"Negotiations using secure multi-party computation"

Mawet, Sophie

Abstract

Secure multi-party computation is a problem where a number of parties want to compute a function of their inputs in a secure way. Security implies correctness of the outputs and privacy of the inputs, even when some parties are cheating. This problem has been at the centre of cryptography research for almost 30 years. However, it is only recently that practical applications have been developed, for example, in auctions, voting systems or data mining. In this vein, this thesis aims to securely solve classical algorithmic problems using multi-party computation techniques, but departs from the traditional focus on problems that admit a simple circuit representation to investigate problems with a richer structure. First, this work presents new sorting algorithms based on a unary representation of integers. These algorithms can be used efficiently as subroutines for applications that make use of the unary representation, for example, in addressing mechanisms. Second, a new procedure to ob...

Document type : *Thèse (Dissertation)*

École polytechnique de Louvain
Institut ICTEAM

Negotiations using Secure Multi-Party Computation

Sophie Mawet

Thèse soutenue en vue de l'obtention du grade de
docteur en sciences de l'ingénieur

Composition du jury:
Prof. Olivier PEREIRA, UCL, Promoteur
Prof. Jean-Pierre RASKIN, UCL, Président
Prof. François-Xavier STANDAERT, UCL, Secrétaire
Prof. Claudia DIAZ, KU Leuven
Prof. Peter RYAN, University of Luxembourg
Prof. Mathieu VAN VYVE, UCL

Louvain-la-Neuve, Belgique, Avril 2015.

Acknowledgements

Coming to the end of my thesis, I am pleased to express my gratitude to all the people who helped me during this period.

First of all, I am very grateful to my supervisor, Professor Olivier Pereira, who gave me the opportunity to carry out my doctoral research in the Crypto Group. His guidance and encouragements were very precious and useful. I am also thankful to my supervising committee, Professor François-Xavier Standaert and Professor Mathieu Van Vyve, for their helpful advice, to the external members of the jury, Professor Claudia Diaz and Professor Peter Ryan, for their careful reading of my work and to Professor Jean-Pierre Raskin for accepting to chair the jury. I sincerely thank all of them for their constructive comments and challenging questions that helped me to improve my thesis.

It was a great opportunity to be part of the Crypto Group during my PhD. My thanks go to all the past and present members of the group who made my research enjoyable. In particular, working next to my office mate, Saloomeh Shariati, was a real pleasure. I am indebted to François Koeune, Christophe Petit and Nicolas Veyrat-Charvillon for sharing their knowledge with me and directing my attention to good references. I also thank Brigitte Dupont for all I have learnt about Linux systems and computer management, my co-authors Abdel Aly and Édouard Cuvelier for our collaboration as well as Sylvie Baudine, Marie Colling and Mathieu Renauld, who helped proof-reading this text.

This thesis would not have been possible without the support of the Fonds National de la Recherche Scientifique. I thank them for funding this research and for their trust. I am also grateful to the Université catholique de Louvain for providing good working conditions and offering a broad education through various trainings and cultural activities, for example.

During this challenging period of PhD, I could count on my family and friends to take my mind off my work. I am specially thankful to Adèle, Élisabeth,

Hélène and Marilyn.

Finally, I would like to dedicate this thesis to my grand parents, my mother and my husband for their kindness and loving support.

Sophie Mawet,
Louvain-la-Neuve, March 2015.

Contents

Notations and Acronyms

Cryptographic Symbols and Notations

$\mathcal{F}_{ABB}$	Arithmetic Black-Box Functionality
$[x]$	Share of x
$\mathbb{Z}_m$	Ring of size m
t	Threshold
$\mathcal{S}$	Simulator
$\mathcal{T}$	Trusted Party
$\mathcal{O}()$	Asymptotic complexity

Acronyms

AES	Advanced Encryption Standard
AKS	Ajtai, Komlós and Szemerédi (sorting network)
FIFO	First In, First Out
GC	Garbled Circuit
GF	Galois Field
HE	Homomorphic Encryption
IT	Information Theory
LU	Lower Upper (factorization)
MPC	Multi-Party Computation
PQ	Priority Queue

SSMF	Single-Source Maximum Flow
SSSP	Single-Source Shortest Path
VIFF	The Virtual Ideal Functionality Framework

Fair Division Notations

N	Number of intervals of the cake
u_m	Player P_1's utilities for the N intervals of the cake
v_m	Player P_2's utilities for the N intervals of the cake
w_m	Player P_3's utilities for the N intervals of the cake
$order$	Players' order for the equitable division (for example, order 1 corresponds to order P_1-P_2-P_3)
$\tilde{\alpha}_1$	Approximation to the first cutting point of the equitable division with the players' order 1. ($\alpha_1 = \frac{\tilde{\alpha_1}}{2^s+l-r}$)
α_1	Exact value of the first cutting point with the players' order 1 $\alpha_1 = \frac{a_{\alpha_1}}{b_{\alpha_1}}$
i	Interval of the first cutting point ($i-1 \leq \alpha_1 < i$)

Graph Notations

$G = (V, E)$	Graph made up of a set of vertices V and a set of egdes E
$\|E\|$	Cardinality of the set E
s	Source vertex
t	Target vertex
P	Predecessor
D	Distance
W	Weight (or cost) of an edge
C	Maximum capacity of an edge
$h(e)$	Head vertex of an edge e
$t(e)$	Tail vertex of an edge e
$\top$	Higher bound

Chapter 1

Introduction

Nowadays, cryptography can be found everywhere. During the last century, cryptography has quickly evolved from an art mainly used by the army to a science used by billions of people. At the same time, the number of applications also exploded: message encryption and authentication, digital signatures, digital cash and electronic voting to cite but a few.

In this thesis, we focus on a particular area of cryptography: secure multi-party computation. Secure MPC is a problem where a number of parties want to compute a function of their inputs in a secure way (Figure 1.1). Security implies correctness of the outputs and privacy of the inputs, even when some parties are cheating. This problem has been at the centre of cryptography research for almost 30 years. However, it is only recently that practical applications have been developed, for example, in auctions, voting systems or data mining.

Following these results, the research question of this thesis is to identify more complex algorithmic problems that can be securely solved using multi-party computation techniques. In other words, the goal of this thesis is to depart from the traditional focus on problems that admit a simple circuit representation to investigate problems with a richer structure.

Section 1.1 goes through the scope and motivation of our research. Section 1.2 details the core contributions of our work and Section 1.3 presents the outline of the thesis.

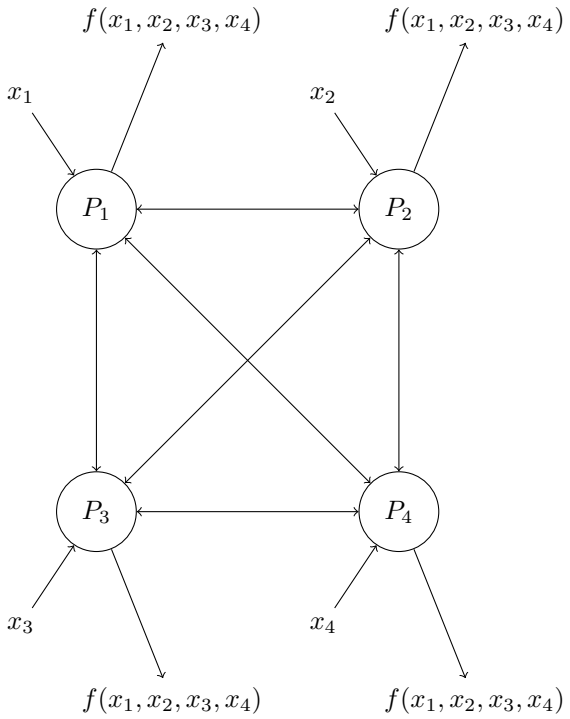


Figure 1.1: Secure multi-party computations between 4 parties.

1.1 Scope and Motivation

In many cases, competing or distrustful parties want to acquire some common information based on private data. A major challenge is to reveal the useful information without leaking anything else. These settings are very common in today's world as illustrated by the wide range of applications hereunder.

- Benchmarking is a typical application of secure multi-party computation. Different companies want to compare their performances but no company is ready to reveal its key figures. For example, large-scale retailers would be interested in comparing their salary costs, hospitals their nosocomial infection rates or farmers their production efficiency.
- In auctions, the auctioneer and the bidders want to know who won the auction as well as the value of the highest bid. However, they may also be interested in keeping the individual bids secret. A private auction can increase the profit for the auctioneer by making bidders give their true valuations of the item. At the same time, this true valuation is kept secret so the bidders do not take the risk of revealing sensitive information that could be misused later.
- In voting systems, the parties want to learn the winner and preserve the privacy of their votes. Voting schemes cannot leak any information except the result of the vote.
- Data mining applications are also promising. Parties with confidential databases want to extract common information without revealing the content of their database. For example, intelligence agencies could share information on potential terrorists.

In today's literature, a lot of work has been done to evaluate a given circuit efficiently. However, it is not enough for many high-level problems. Our work is independent from this research direction. In this thesis, we investigate secure multi-party computation techniques to solve classical algorithmic problems securely while avoiding a combinatorial explosion. The solutions we propose are not exclusively dedicated to a specific application. For example, our sorting algorithms could be used to compute the winner of an auction as well as to benchmark the well-being at work in different sectors of a large company. In summary, our MPC algorithms can be viewed as a library of building blocks for a wide range of secure applications.

1.2 Main Contributions

In this thesis, we investigate sorting, game theory and graph theory problems, which cannot be solved securely using traditional circuit-based approaches. Our main contributions to new secure algorithms are listed below.

- We present new sorting algorithms based on a unary representation of integers. These algorithms can be used efficiently as subroutines for applications that make use of the unary representation, for example, in addressing mechanisms. We benchmark these algorithms with a secure implementation of a sorting network. The design of these algorithms strongly differs from standard techniques such as Quicksort, Shellsort or sorting networks.

- We provide a new procedure to obtain a fair division of a heterogeneous resource between competing parties. This procedure takes advantage of the secure setting (the preferences of the parties are kept secret). The procedure does not have a counterpart in game theory and enables reaching an equilibrium that dominates those that were previously known in unmediated procedures. This result was presented at WISSec 2010 [1].

- We develop the first secure single-source shortest path and maximum flow algorithms. None of the known algorithms for solving these problems can be described as circuits, as needed for an immediate transposition to a secure setting. Depending on the setting, our algorithms raise intriguing questions in terms of asymptotic complexity, compared to their traditional counterpart (Bellman-Ford, Dijkstra, Edmonds-Karp,...). We compare the different versions and propose implementation prototypes. This result was presented at Financial Cryptography 2013 [2].

The algorithmic problems we solved have been studied thoroughly, independently of any security concern. We highlight fundamental problems that arise when they need to be solved securely. Three challenges seem particularly important.

- *Complexity Gaps.* This thesis analyses the complexity gaps between the traditional algorithms and their secure counterparts. A multitude of criteria come into play to determine these gaps. The data to keep confidential, the structure of the problem and the type of primitives used are as many central elements. For example, the best solution to achieve a fair and secure division of a resource (plot of land, house chores,...) is to design a completely new algorithm. Conversely, it is possible to translate sorting networks quite directly into a secure version.

  There is very little research about these complexity gaps. Our work

highlights little known problems and gives rise to many open questions. For example, it is not clear whether these complexity gaps are inherent to a particular problem or if there is a way to prove a lower bound on these gaps.

- *Leakage by data structure and execution flow.* Most traditional efficient algorithms have an execution flow that depends on the data that are manipulated: control flow will happen as a function of data that must be kept secret, as the result of branching or loop exit conditions for instance. Therefore, new algorithms are needed in order to prevent undesirable information leakages without causing combinatorial explosions that trivial solutions would produce.
- *Different efficiency metrics.* The traditional complexity metrics do not transpose to secure computation. For instance, it is typically more expensive, by more than two orders of magnitude, to compare values than to multiply them. As a result, some shortest path algorithms with higher complexity become faster for most problems with practical size, due to the fact they require fewer comparisons than multiplications. This motivates substantial departures from traditional approaches to reach practical/optimal solutions.

These lines of research are expected to bring algorithmic advances that are crucial towards the practical use of privacy-preserving algorithms and, as a consequence of the removal of privacy concerns, a more effective collaboration between competing entities on various markets.

1.3 Organization of the thesis

The thesis is organized in three main parts. Part 1 consists of Chapter 2 and 3. It is a preliminary part and presents material used all along the thesis. For example, it details the setting, some notations and a specific representation for integers. Part 2 consists of Chapters 4, 5 and 6. It is the core part of our work and includes our contributions on different multi-party computation protocols. Part 3 consists of Chapter 7. It draws the conclusions of our research.

Chapter 2 presents a short introduction to the multi-party computation techniques used in the thesis. We present basics like the type of adversaries, the model of communications and the fundamental impossibility results for threshold adversaries. Then, we describe the outlines of a secure multi-party protocol, from secret sharing to result reconstruction. We also give an insight into the

various practical applications and the current frameworks before presenting the setting and notations we will use throughout the thesis. Appendix A provides a summary of well-known MPC primitives.

Chapter 3 introduces a different way to represent the shared values. The shared values are represented as unary counters. We describe the definition of unary counters as well as basic operations like indexing, updating a vector at a shared position or incrementing a shared unary index. We also detail interesting applications of the unary representation. We try to give a quite complete overview of the strengths and weaknesses of this approach because it will be used in several parts of the thesis.

Chapter 4 presents secure sorting protocols that can be used as such or as building blocks for secure applications. We explore two approaches. In the first one, we used sorting networks. Thanks to this technique, it is quite easy to obtain a secure protocol without any asymptotic overhead. The second approach uses the unary representation of the integers in order to sort them. This chapter also illustrates the fundamental problems that arise when classical algorithms have to be solved securely. Our secure protocols rely on previous works on MPC and on traditional sorting algorithms.

Chapter 5 presents secure solutions to a game-theoretic problem: the cake-cutting problem. Our cryptographic solutions address some important shortcomings of traditional game-theoretic procedures. We describe the modelling of the cake-cutting problem and detail the secure cake-cutting protocol. We also give implementation results. Appendix B provides a short outline on the standard definitions of game theory. Appendix C illustrates these definitions on the cake-cutting problem.

Chapter 6 presents a way to securely solve simple combinatorial graph problems: the single-source shortest path and the maximum flow problems. Our protocols can be used as such or as building blocks for more complex secure applications. It also presents a variant of our single-source shortest path algorithm. We use a priority queue to store the vertices. Thanks to this priority queue, we do not need to compute a minimum at each iteration.

Chapter 7 presents our conclusion and discusses the perspectives for future researches.

Part I

Preliminaries

Chapter 2

Background

Secure multi-party computation – the problem of jointly evaluating a function on a set of secret inputs without leaking anything but the output of the function – has been at the centre of cryptography research for almost 30 years. A first series of foundational works demonstrates the possibility to evaluate any function in various models, the function being described as a circuit [3],[4],[5],[6].

Then, the attention focused largely on building solutions for the evaluation of functions of specific interest, leading to secure and efficient protocols for auctions, voting, benchmarking, data mining, face recognition or AES evaluation, to mention only a few. The common point between all these applications is that entities are reluctant to share their private data.

The goal of this chapter is to present a short introduction to the multi-party computation (MPC) techniques used in the thesis as well as to locate our results among the existing literature. Background about general cryptography can be found in standard textbooks [7], [8], [9] and detailed background about secure multi-party computation can be found in widely available articles [10], [11].

Section 2.1 focuses on MPC basics: the types of adversaries, the models of communication and the impossibility results for threshold adversaries. Section 2.2 describes the outlines of a secure multi-party protocol: the secret sharing, the secure multi-party computation and the result reconstruction. It also highlights the complexity difference that exists between a secure addition and a secure multiplication. Section 2.4 describes the well-studied practical applications of MPC and presents the main current MPC frameworks: Fairplay, Sharemind, SEPIA, TASTY, VIFF and SCAPI. Finally, Section 2.5 presents the setting and notations used throughout the thesis.

2.1 MPC Basics

Secure multi-party computation is the problem of k players who want to compute an agreed function of their inputs in a secure way. Security implies correctness of the outputs and privacy of the inputs, even when some parties are cheating. In concrete terms, there are $k \geq 2$ players P_1,...,P_k, where P_i knows his input x_i. The players want to compute $f(x_1, ..., x_k) = (y_1, ..., y_k)$ so that P_i learns the output y_i and nothing more, except for information that can be deduced from (x_i, y_i). The expression $f(x_1, ..., x_k) = y$ means that all outputs are the same.

Secure multi-party computation can alternatively and more generally be seen as the problem of performing a task among a set of players [12]. The task is specified by involving a trusted party and the goal of the protocol is to replace the need for the trusted party. In other words, the functionality of the trusted party is shared among the players.

Yao's millionaire's problem is a classical illustration of the use of MPC. Two millionaires want to know who is the richest without having to reveal their respective wealth. They compute the function $f(x_1, x_2) = x_1 <^? x_2$ securely where the two inputs x_1 and x_2 stand for the number of millions each of them owns. When the first millionaire is poorer than the second one, the value of the function $f(x_1, x_2)$ is 1, otherwise it is 0. The secure computation of the function by the two millionaires should not reveal anything more about their fortune.

Figure 2.1 illustrates the security definition of a multi-party protocol. The scheme is secure if a player (here P_1) cannot distinguish between the situation where he is actually exchanging information with the other players and the situation where he is exchanging information with a trusted party ($\mathcal{T}$) with the intervention of a simulator ($\mathcal{S}$).

2.1.1 Adversaries

The adversary $\mathcal{A}$ stands for the set of cheating players. In other words, an adversary may corrupt a subset of players. Once corrupted, a player gives the adversary his entire history, i.e., the complete information on all the actions and messages he has received so far.

There are different kinds of corruptions. *Passive corruption* means that the adversary can read all the data of the corrupted players but he cannot modify

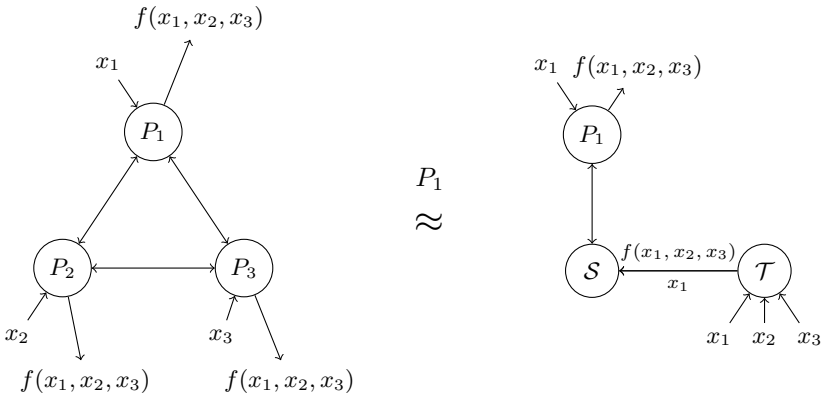


Figure 2.1: The scheme is secure if P1 cannot distinguish between the two situations.

their behaviour, i.e., players still execute the protocol correctly. Passive corruption is also called the honest-but-curious model or the semi-honest model. On the other hand, *active corruption* means that the adversary takes full control of the corrupted players, i.e., players may deviate arbitrarily from the protocol. Active corruption is also called the malicious model.

Static corruption signifies that the subset of corrupted players is fixed in advance whereas *adaptive corruption* signifies that new players can be corrupted during the protocol execution.

2.1.2 Models of Communication

The two basic models of communication are the cryptographic and the information theoretic models.

The *cryptographic model* was introduced by Yao [13] and Goldreich, Micali, and Wigderson [4]. In the cryptographic model, communication channels are supposed to be authenticated but insecure: the adversary has access to all the messages sent but he cannot modify them. Security can thus only be guaranteed in a cryptographic sense, i.e., assuming that the adversary cannot solve some computational problem.

The *information theoretic model* was first called the non-cryptographic model and was introduced by Ben-Or, Goldwasser and Wigderson [5] and Chaum, Crépeau and Damgård [6]. In the information-theoretic (I.T.) model, communication channels are supposed to be pairwise secure, i.e, the adversary gets no

information at all about messages exchanged between honest players. Security can then be guaranteed even when the adversary has unbounded computing power.

Both models have their own advantages and drawbacks. In the cryptographic model, data is encrypted using public-key algorithms. This approach implies the use of far larger integers than the data we are using, hence an important cost in terms of efficiency. However, this model is interesting when a large number of parties wish to compute a quite easy function of their inputs. This is typically the case in voting systems or in auctions. The information theoretic model is based on a sharing of the data rather than a sharing of the keys. Because of the need to distribute independently shared versions of the secret data, this model is more relevant for applications with a small number of parties. This is typically the case for negotiation problems. The I.T. model allows us to work with smaller integers. Contrary to the cryptographic model, the integers are of the same size as the data used. For these reasons, the information theoretic model seems more suitable for our problems.

In this work, communication is assumed to be synchronous in the I.T. model. A protocol proceeds in rounds: in every round, each player may send a message to each other player, and all the messages are delivered before the next round begins. In an asynchronous model of communication, there is no guarantee on message delivery or bounds on transit time. Problems can thus only be solved in a strictly weaker sense.

2.1.3 Impossibility Results for Threshold Adversaries

A protocol cannot be secure if *any* subset of the k players can be corrupted. Limitations must be specified on the subsets the adversary can corrupt. If the adversary is allowed to corrupt all the subsets of the parties of size at most t for some $t < k$, then it is called a threshold adversary and t is called the threshold.

In the cryptographic model, security with a computationally bounded threshold adversary is possible if at most $t < k/2$ of the players are corrupted. The same result holds for active and passive corruption [4].

In the information theoretic model, unconditional security is possible if at most $t < k/3$ of the players are actively corrupted. For passive corruption, the threshold is $t < k/2$. The bound $t < k/3$ for active corruption can be reduced to $t < k/2$, assuming the existence of a broadcast channel [4], [6].

2.2 Outlines of a Secure MPC Protocol

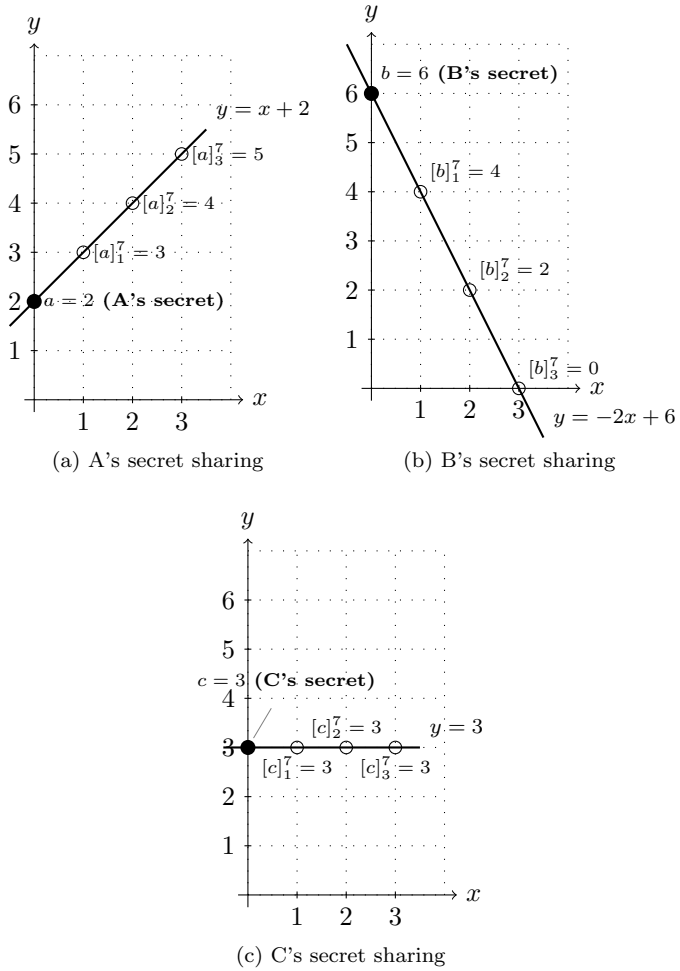
Secure multi-party computation can be divided typically into three phases. First, each player P_i shares his input x_i secretly between the k players (including himself). Now, all the players have a secret share of the input x_i of each other player. Second, each player runs a protocol with, as inputs, the k shares he has received. The goal is to compute the function $y = f(x_1, \ldots, x_k)$ securely without a trusted party. Each player's output is a share of the secret value y. Third, the shares can be used for further computation or revealed to the players who can then reconstruct y.

For example, suppose that k players want to compute the function $y = x_1 + \ldots + x_k$ securely in the I.T. scenario with a passive adversary. Each player P_i can use Shamir's scheme [14] to share his secret $x_i \in \mathbb{Z}_q$, where $\mathbb{Z}_q = \{x \in \mathbb{Z} \,|\, 0 \leq x \leq q-1\}$, $q > k$ and q is a prime. Player P_i chooses random $c_{mi} \in \mathbb{Z}_q$ for $m = 1, ..., t$, and sets $[x_i]_j^q = x_i + \sum_{m=1}^{t} c_{mi} j^m \mod q$. Acting like this, P_i has the guarantee that less than $t+1$ shares $[x_i]_j^q$ do not reveal anything about his secret x_i. Player P_i sends $[x_i]_j^q$ to player P_j. Shamir's scheme is linear, so addition is performed by having all players locally adding their shares. Player P_i gets his share $[y]_i^q$ of the secret y by computing $[x_1]_i^q + \ldots + [x_k]_i^q$. Any set of t players can reconstruct the secret value y by interpolating their shares.

2.2.1 Secret Sharing

In concrete terms, suppose three players (A, B and C) want to compute the sum of their secrets (a, b and c) without having to reveal them. Let $\mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$, $a = 2$, $b = 6$ and $c = 3$. The sum is $s = 11 \equiv 4 \pmod 7$. One player at the most can be corrupted ($t = 1$ because $t < k/2$), the secrets are thus shared through polynomials of degree 1. Each player selects one random element in $\mathbb{Z}_7$ for the coefficient of the polynomial (the independent term is the secret). Players A, B and C choose $\alpha = 1$, $\beta = 5$ and $\gamma = 0$ respectively. Each player now has a polynomial to share his secret. For example, player A sets $[a]_i^q = a + \alpha \cdot i^1 \mod q$.

Figure 2.2 illustrates the secret sharing of each player while Table 2.1 describes the corresponding computations.



(a) A's secret sharing

(b) B's secret sharing

(c) C's secret sharing

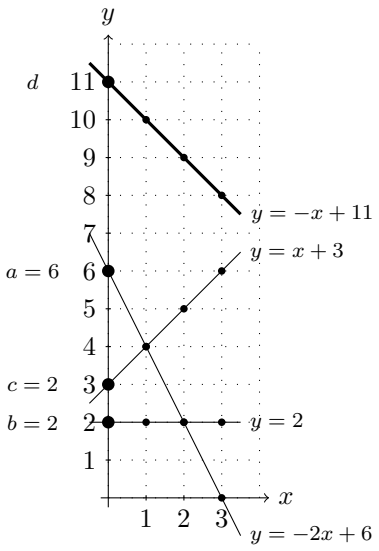
Figure 2.2: Secret sharing.

	Shares of $a = 2$ sent from A	Shares of $b = 6$ sent from B	Shares of $c = 3$ sent from C
to A	$[a]^7_1 = 2 + 1 \cdot 1 = \mathbf{3}$	$[b]^7_1 = 6 + 5 \cdot 1 = 11 \equiv \mathbf{4}$	$[c]^7_1 = 3 + 0 \cdot 1 = \mathbf{3}$
to B	$[a]^7_2 = 2 + 1 \cdot 2 = \mathbf{4}$	$[b]^7_2 = 6 + 5 \cdot 2 = 16 \equiv \mathbf{2}$	$[c]^7_2 = 3 + 0 \cdot 2 = \mathbf{3}$
to C	$[a]^7_3 = 2 + 1 \cdot 3 = \mathbf{5}$	$[b]^7_3 = 6 + 5 \cdot 3 = 21 \equiv \mathbf{0}$	$[c]^7_3 = 3 + 0 \cdot 3 = \mathbf{3}$

Table 2.1: Computation and exchange of shares between the players.

2.2.2 Secure Computation and Secret Reconstruction

After this first step, the secret sharing, the second step deals with the multi-party computation strictly speaking. Each player performs the same computation (here a sum) on the shares he received. Secure addition can be achieved locally. Figure 2.3 illustrates the local sum of each player. For the sake of clarity, the result is not represented modulo 7 (as it should be).



		A	B	C
a	6	4	2	0
b	2	2	2	2
c	3	4	5	6
$d = a + b + c$	11	10	9	8
$d \mod 7$	4	3	2	1

Figure 2.3: Sum of three shared values.

The corresponding computations of A,B and C are detailed here. A, B and C compute $[d]_1^7 = [a]_1^7 + [b]_1^7 + [c]_1^7 = 10 \equiv 3 \pmod 7$, $[d]_2^7 = [a]_2^7 + [b]_2^7 + [c]_2^7 = 9 \equiv 2 \pmod 7$ and $[d]_3^7 = [a]_3^7 + [b]_3^7 + [c]_3^7 = 8 \equiv 1 \pmod 7$ to have a polynomial share of $d = a + b + c$. Two players (at least) compute the secret $d = 4$ by interpolating their shares ($[d]_1^7 = 3$, $[d]_2^7 = 2$ and $[d]_3^7 = 1$).

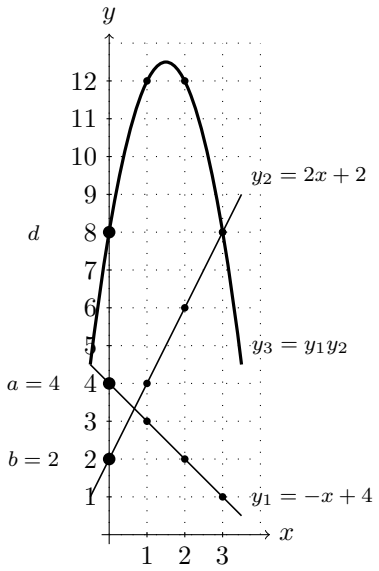
2.2.3 A More Complex Example

Secure addition of shared secrets is local and easy but other functions can be more complicated to evaluate securely. Multiplication, for example, cannot be performed locally. Multiplication uses a re-sharing step that requires one round of communication between the players. Let us assume that a and b are

shared polynomially over a large field between A, B and C. As it can be seen on Figure 2.4, the local multiplication leads to a 2-degree polynomial, whose independent term is $a \cdot b$. This polynomial has to be replaced by a random polynomial of degree 1 with the same independent term. It will be achieved thanks to a re-sharing step.

The three steps to multiply two shared values $[a]_i^q$ and $[b]_i^q$ are summarized below. A detailed proof is given by Gennaro *et al.* [15].

- Local multiplication: $[d]_i^q = [a]_i^q[b]_i^q$.
- Re-sharing of $[d]_i^q$ by choosing a random polynomial $h_i(x)$ of degree 1 so that $h_i(0) = [d]_i^q$.
- Recombination by computing the linear combination $H(j) = \sum_{i=1}^{3} \lambda_i h_i(j)$. The Lagrange coefficients λ_i are public and independent of the shares.



		A	B	C
a	4	3	2	1
b	2	4	6	8
$d = ab$	8	12	12	8
d_1	12	10	8	6
d_2	12	9	6	3
d_3	8	9	10	11
$e = 3d_1 - 3d_2 + d_3$	8	12	16	20

Figure 2.4: Multiplication of two shared values.

As we can see, secure addition and secure multiplication are already different in terms of complexity and communication. Other functions, like comparison, involve a lot more rounds and have a higher complexity (see Appendix A). These differences of complexity are one of the big challenges when developing secure applications. The most efficient algorithm might not be the best one to use in MPC.

2.3 A Related Tool: Oblivious Memories

Oblivious memories (ORAM) [16], [17], [18], [19] allow outsourcing some data on a remote server and accessing it in a way that makes it infeasible to know what specific piece of data has been accessed. Oblivious RAM might speed up secure multi-party computation when they involve large data, for example in data mining. However, it comes with a non-negligible computational overhead, $\mathcal{O}(\log^2 n)$, in the best solutions. This overhead can be fairly important given the relatively small representations that are often at stake in negotiation problems.

We also observe that the techniques used there are fundamentally probabilistic, which is quite different from the approach that we use, which seeks for obliviousness and has deterministic execution patterns. Oblivious RAM or similar oblivious data structures could be a very useful tool in specific cases, as a complement of the techniques we use. However, it does not seem to help for branching which is the problem we encounter the most in our negotiation problems.

2.4 MPC in Practice

This section presents some applications of MPC and gives the current implementations of MPC protocols. Some frameworks are tailored for specific applications (data mining, for example). The main features of each implementation are summarized in Table 2.2.

2.4.1 Applications

Since the first fundamental feasibility results of MPC more than 30 years ago, many practical applications have been proposed. The MPC techniques are particularly useful for applications involving different participants with conflicting interests who want to keep their data confidential. This is the case for numerous varied applications.

Bogetoft *et al.* give an implementation of secure auctions for practical real-world problems [20]. They address the problem of double auctions of a single divisible commodity with multiple sellers and buyers. The disclosure of an individual bid may be of great interest for the other buyers, who can use the revealed information to adapt their strategy, not only for the current auction but also for upcoming ones. This result led to the first real-world MPC application [21]. In 2008, MPC were used to determine the annual Danish market price of sugar

beets. The national auction took place with about 1200 farmers and lasted about 30 minutes. In 2009, Miltersen *et al.* proposed a rational cryptographic protocol for a single item auction [22].

In 1996, Cramer *et al.* presented an efficient voting scheme that satisfies universal verifiability, privacy and robustness. The computational and communication complexity are essentially linear, instead of quadratic as for previous schemes [23]. In 2007, Clarkson *et al.* presented Civitas, the first electronic voting system that is coercion-resistant, universally verifiable and suitable for remote voting [24]. Civitas uses MPC techniques for computing the tally. In 2009, the rector of the Université catholique de Louvain was elected using an open-audit system [25]. In 2011, the Norwegian government ran a trial of internet remote voting. During the local government elections, electors in 10 municipalities voted from home with their own computers. The voting system is based on ElGamal encryption of ballots and a mix-net for decryption [26], [27].

Branching programs are commonly used to model diagnostic and classification algorithms with applications in areas such as health care, fault diagnostic or benchmarking. Barni *et al.* present efficient privacy-protecting protocols for remote evaluation of such algorithms. They apply their protocols to the secure classification of medical ElectroCardioGram (ECG) signals [28].

Lindell and Pinkas address the problem of secure data mining. They propose a protocol based on the ID3 algorithm that is more efficient than generic solutions in terms of number of rounds of communication and bandwidth [29], [30]. Lindell and Pinkas provide a survey of the basic paradigms and notions of secure multi-party computation used for secure data mining [31]. As detailed in Section 2.4.2, Sharemind is a dedicated framework for data mining applications. Bogdanov *et al.* developed high-performance secure multi-party computation for data mining applications [32]. Sharemind was used for financial reporting in a consortium [33], for large-scale genome-wide association studies [34] and for studies on linked databases [35].

Automatic recognition of human faces is another topic that raises important privacy issues, for example, when a client searches privately for a specific face image in the database of a server. Erkin *et al.* are the first to propose a recognition system that hides the biometrics and the result from the server performing the computation [36]. Sadeghi, Schneider and Wehrenberg also propose a secure face recognition scheme and give implementation results that show the practicality of their solution even for large databases [37].

Similar techniques have been used for AES evaluation [38], [39], [40]. These last results show the fast progress in generic MPC. In 2009, Pinkas *et al.* provided

the first implementation of MPC for 2 parties with active security. Their implementation was able to evaluate a circuit of about $3 \cdot 10^4$ gates in about 10^3 seconds [38]. In 2012, Nielsen *et al.* were able to evaluate the same circuit in less than 5 seconds [40].

2.4.2 Frameworks

In the last years, various MPC frameworks have been implemented. They were often designed with a specific application in mind. SEPIA and VIFF are based on Shamir's secret sharing scheme and are the most general multi-party frameworks. Sharemind is a 3-party framework dedicated to computations on large databases. Fairplay and TASTY are 2-party frameworks and require therefore computational security. These frameworks are based either on homomorphic encryption (HE) or garbled circuits (GC). Homomorphic encryption is computationally expensive, while garbled circuits require generating a circuit computing the function. We describe hereunder the most well-known frameworks.

Fairplay (`www.cs.huji.ac.il/project/Fairplay`) is a general-purpose framework for MPC [41]. It uses a high-level function description language. Functions are computed using Yao's protocol for secure evaluations of boolean circuits. Fairplay was designed for two-party computation but has been extended for multi-party computation: FairplayMP [42].

Sharemind (`sharemind.cyber.ee/research`) is a framework for secure data mining applications [43]. It uses a special choice of arithmetic: an additive secret sharing scheme over the ring $\mathbb{Z}_{2^{32}}$. SecreC is a secure programming language designed for data mining tools.

SEPIA - SEcurity through Private Information Aggregation (`sepia.ee.ethz.ch`) is a Java library for MPC using Shamir's secret sharing [44], [45]. It is tailored for network security and monitoring applications (the basic operations are optimized for large numbers of parallel invocations).

TASTY - Tool for Automating Secure Two-partY computation is a general-purpose tool for secure computation between exactly 2 parties [46]. It uses combinations of garbled circuits and homomorphic encryption techniques.

VIFF - The Virtual Ideal Functionality Framework (`http://viff.dk`) is a high-level framework for asynchronous multi-party computation [47], [48]. It is implemented in Python and exploits the Twisted library for the asynchronous communications. It uses Shamir's secret sharing and the arithmetic operations include secure comparison. VIFF proposes protocols for passive and active security.

	Implementation language	Building techniques	Number of participants
Fairplay / FairplayMP	SFDL (Java)	Garbled circuits	2 parties / 3 or more parties
Sharemind	SecreC (C++)	Additive secret sharing	exactly 3 parties
SEPIA	Java	Shamir's secret sharing	3 or more parties
TASTY	Tastyl (based on Python)	GC and HE	2 parties
VIFF	Python	Shamir's secret sharing	3 or more parties
SCAPI	Java	Garbled circuits	2 parties and multi parties
Wysteria	Wysteria (based on OCaml)	Secret sharing	2 parties and multi parties

Table 2.2: Comparison of the current MPC frameworks.

SCAPI - The Secure Computation Application Programming Interface (`crypto.biu.ac.il/scapi`) will be a Java library designed for secure computation [49]. Still under development, it is meant to be a general infrastructure and does not target any specific protocol.

Wysteria is a high-level programming language for generic, mixed-mode multi-party computations [50]. The novelty of this programming language is to propose two computation modes: a secure mode to perform synchronous MPC and a parallel mode to perform local and private computations in parallel.

2.5 Security Assumptions and Notations

We build our protocols on top of an ideal functionality: the arithmetic black-box functionality $\mathcal{F}_{ABB}$ of Damgård and Nielsen [51] whose definition captures the properties we need. This functionality allows n parties to store elements of a ring $\mathbb{Z}_m$ securely, to perform the ring operations of addition and multiplication on these elements repeatedly, and to open the result of the computation when needed. Following Toft [52], we consider a slightly extended and abstracted version of this functionality that offers the possibility to perform secure compa-

rison and consider any possible ring. So, storing, opening, adding, multiplying and comparing will be the only secure operations on which our protocols will rely.

Addition and multiplication by a public value are costless due to the linearity of the primitives. Communication complexity will mostly be measured by the number of secure multiplications performed. The number of secure comparisons will also be used to explain the difference of efficiency observed in practice.

Following the tradition, we will write $[x]$ to address the version of x stored securely by $\mathcal{F}_{ABB}$ and $[\mathbf{A}]$ to address a secret shared array $[\mathbf{A}] = ([a_1], [a_2], ..., [a_n])$. Traditional indexing is written $[\mathbf{A}](i)$. Oblivious indexing is written $[\mathbf{A}]([\mathbf{i}])$ because it is obtained by using a shared unary index of the same size as the array (see Chapter 3). We will denote the secure arithmetic operations on secret values in the natural way, e.g., $[z] \leftarrow [x] + [y]$ for the addition of two secrets. The actual protocol implementing these operations depends on the details of the realization of this functionality. Numerous MPC schemes can be used for that purpose depending on the security model that is appropriate. For example, the schemes of Goldreich *et al.* [4] and Chaum *et al.* [6] or, for more recent approaches those of Bendlin *et al.* [53] and Damgård *et al.* [54], [55].

In this thesis, we work in the honest-but-curious model. The adversary can control up to $n_c = \lfloor (n-1)/2 \rfloor = 1$ player. The corrupted player follows the protocol but tries to learn as much as possible about the inputs of the other parties.

Chapter 3

Unary Representation

In this chapter, we introduce a different way to represent integer values: we represent them as unary counters. This representation will be used for a specific secure sorting algorithm in Chapter 4 and for a secure single-source shortest path algorithm in Chapter 6. Chapter 5 does not use a unary notation explicitly but uses an idea of the same flavour.

The goal of this chapter is to introduce some basics about this notation. We illustrate the main advantages and drawbacks of dealing with a unary representation. Using a unary notation helps obliviously performing some important operations like indexing. However, it often increases the complexity.

Section 3.1 defines unary counters and shows how useful they are in a secure setting. Section 3.2 describes some basic operations with unary counters like updating an element at a shared position in an array or computing the minimum of a list along with its index. Finally, Section 3.3 gives some interesting applications of unary counters. For example, we present a knapsack algorithm where the unary notation does not come with an asymptotic overhead.

3.1 Unary Counters

A unary counter of i is an array consisting of all 0's except for the i^{th} position that is 1. The length of the array depends on the size of the domain that is the size of the interval. For example, the unary counter of the integer value $i = 3$ in an interval of size $n = 5$ would be $\mathbf{i} = (0 \quad 0 \quad 0 \quad 1 \quad 0)$. In a shared form, it is written as $[\mathbf{i}] = ([0] \quad [0] \quad [0] \quad [1] \quad [0])$.

The way a unary counter is constructed from a shared integer value has been described by Reistad and Toft [56]. It is similar to the evaluation of a symmetric Boolean function described by Damgård *et al.* [57]. The construction uses n public Lagrange polynomials of degree $n-1$ (with n the size of the unary representation). For example with $n=5$, the first polynomial interpolates the points $((0,1),(1,0),(2,0),(3,0),(4,0))$, the second polynomial the points $((0,0),(1,1),(2,0),(3,0),(4,0))$ and so on. As these polynomials are public, we only need to compute the powers of the integer value i: $i,...,i^{n-1}$ using a prefix-product. After this, it is costless to evaluate each Lagrange polynomial in i. The final complexity for constructing a unary counter from a shared integer value is equivalent to a prefix-product of $n-1$ terms.

Unary counters are useful to access an element in an array obliviously. We only need to express the index i as a unary counter. Suppose we have an array $[\mathbf{A}]$ of size n, the shared index is written $[\mathbf{i}]$ and is also of size n. Then, accessing an element at a shared index position $[\mathbf{i}]$ is written $[\mathbf{A}]([\mathbf{i}])$ and corresponds to the computation of a dot product:

$$[\mathbf{A}]([\mathbf{i}]) = \sum_{j=0}^{n-1} [\mathbf{A}](j) \cdot [\mathbf{i}](j)$$

Accessing an element with a shared index requires n secure multiplications (linear overhead), that may be performed in parallel.

Thanks to the unary representation, we can perform an equality test with a dot product. It costs again n multiplications. However, it is convenient for small n because a product costs less than an equality test.

$$[x] = ([3] \stackrel{?}{=} [3]) \qquad \rightarrow \qquad [x] = \begin{pmatrix}[0] & [0] & [0] & [1] & [0]\end{pmatrix} \cdot \begin{pmatrix}[0]\\[0]\\[0]\\[1]\\[0]\end{pmatrix}$$

Finally, the integer value of i may be computed again as

$$i = \sum_{j=0}^{n-1} [\mathbf{i}](j) \cdot j.$$

The multiplications between the shared values $[\mathbf{i}](j)$ and the public value j are all local and have thus no cost.

Unary counters have been used, amongst others, by Toft to solve linear programs [58], by Launchbury *et al.* to develop secure lookup tables [59] and by Aly *et al.* to solve combinatorial graph problems securely [2].

3.2 Unary Operations

Let us describe some basic examples of operations on vectors of shared values. The update of a shared value with a public index is trivial and has no overhead compared to the non-secure version.

However, the update of a shared value with a private index (Protocol 1) has a linear overhead. Indeed, to update an element of a vector privately, we have to pass on the whole vector to avoid leaking information. This function uses therefore n multiplications with n equal to the length of the vector.

Protocol 1: Update an element in a shared list and at a shared position.

Input: A list $[\mathbf{A}]$ of length n, a shared index $[\mathbf{i}]$ of length n, a shared value $[x]$.
Output: The list $[\mathbf{A}]$ with the update $[\mathbf{A}]([\mathbf{i}]) = [x]$.

for $j \leftarrow 1$ **to** n **do**
  $[\mathbf{A}](j) \leftarrow [\mathbf{A}](j) + [\mathbf{i}](j) \cdot ([x] - [\mathbf{A}](j))$;
end
return $[\mathbf{A}]$;

Protocol 2 obliviously increments a shared index: the index is only incremented by 1 if `inc` $= 1$. For example, update-index($[\mathbf{i}]$=([0],[1],[0],[0],[0]), inc=[1]) returns ([0],[0],[1],[0],[0]) and `update_index`($[\mathbf{i}] = ([0], [1], [0], [0], [0]), \texttt{inc} = [0]$) returns $([0], [1], [0], [0], [0])$. The method uses n multiplications where n is the length of the vector $[\mathbf{i}]$. It also has a linear overhead compared to a non-secure version.

Protocol 2: Increment a shared index if the increment is 1.

Input: A shared index $[\mathbf{i}]$ and a shared increment [inc].
Output: The index $[\mathbf{i}]$ is incremented if inc $= [1]$.

for $j \leftarrow n - 1$ **to** 1 **do**
  $[\mathbf{i}](j) \leftarrow [\mathbf{i}](j)+$ [inc] $\cdot([\mathbf{i}](j-1) - [\mathbf{i}](j))$;
end
[$\mathbf{i}$](0) $\leftarrow$ (1-[inc]) $\cdot$[$\mathbf{i}$](0);
return $[\mathbf{i}]$;

Protocol 3 has been introduced by Toft to obtain the minimal value out of a vector of shared values [58]. It securely computes a share of the minimal value, [min], along with a share of its index, [**i**]. The protocol uses $\mathcal{O}(n)$ comparisons and multiplications. Its overall round complexity is $\mathcal{O}(\log(n))$ rounds.

Protocol 3: binarymin. Compute the minimal element in $\mathcal{O}(log(n))$ rounds and $\mathcal{O}(n)$ comparisons.

Input: A list [**A**] of length n with n a power of 2.
Output: The minimum [min] of [**A**] along with its index [**i**].

if $n = 1$ **then**
  [min] $\leftarrow$ [**A**](0);
  [**i**] $\leftarrow$ [1];
  return([min], [**i**]);
else
  for $j \leftarrow 0$ **to** $n/2 - 1$ **do**
    [**B**](j) $\leftarrow$ [**A**]($2 \cdot j$) $<$ [**A**]($2 \cdot j + 1$);
    [**A'**](j) $\leftarrow$ [**B**](j) $\cdot$ ([**A**]($2 \cdot j$) $-$ [**A**]($2 \cdot j + 1$)) $+$ [**A**]($2 \cdot j + 1$);
  end
  ([min], [**i'**]) $\leftarrow$ `binarymin`([A']);
  for $j \leftarrow 0$ **to** $n/2 - 1$ **do**
    [**i**]($2 \cdot j$) $\leftarrow$ [**B**](j) $\cdot$ [**i'**](j);
    [**i**]($2 \cdot j + 1$) $\leftarrow$ ($1 -$ [**B**](j)) $\cdot$ [**i'**](j);
  end
  return([min], [**i**]);
end

These protocols will be used in various sorting and graph protocols.

3.3 Applications

Unary representations have important advantages. For example, it enables checking the number of occurrences of a given value x in a private array easily and obliviously. If x is public it can even be computed without any communication between the parties. This is achieved through a dot product between the private array and the unary index of x. It gives a share of 0 if x is not in the array and a share of the number of occurrences otherwise. The cost is very low: n parallel multiplications. These multiplications are local if x is public.

Some NP-complete problems, like the knapsack problem can be solved thanks to a pseudo-polynomial algorithm [60],[61],[62]. These algorithms are polynomial

in the unary size of one variable. For example, Protocol 4 describes a pseudo-polynomial 0/1 knapsack with dynamic programming. In this example, the list $[\mathbf{V}]$ of the values of the n goods is private while the list W of the weights of the n goods is public. As we have a public structure, we do not need to hide the access pattern. This setting makes sense in the case of a public budget where parties support different projects with a known cost but wish to keep secret the valuation they give to their project. Interestingly, this pseudo-polynomial time algorithm already uses a table $\mathbf{M}$ whose length corresponds to the knapsack weight. In this case, no overhead occurs using a unary notation.

Protocol 4: Pseudo-polynomial 0/1 knapsack with dynamic programming.

Input: A private list $[\mathbf{V}]$ of the values of the n goods, a public list W of the weights of the n goods.
Output: The table $[\mathbf{M}]$.

```
1 for i ← 1 to n do
2 |   for j ← 0 to W do
3 |   |   if W(i) ≤ j then
4 |   |   |   [M](i, j) = max([M](i − 1, j), [M](i − 1, j − W(i)) + [V](i));
5 |   |   else
6 |   |   |   [M](i, j) = [M](i − 1, j);
7 |   |   end
8 |   end
9 end
```

Part II

Secure MPC Applications

Chapter 4

Secure Multi-Party Sorting Algorithms

Sorting is a well-studied, yet fundamental problem in algorithms. It can be used in numerous straightforward applications like organizing a library or displaying the list of registered voters to an election but it is also a core subroutine for many algorithms and non-obvious applications, for example, in supply-chain management and graph theory.

The goal of this chapter is to present secure multi-party sorting protocols that can be used as such or as building blocks for secure applications as well as to illustrate the fundamental problems that arise when classical algorithms have to be solved securely. Our secure protocols relies on previous work on MPC and on traditional sorting algorithms.

Section 4.1 describes our contributions and the related works. Section 4.2 introduces sorting networks and presents the way we used them to achieve secure sorting. The implementation prototype of the Odd-even merge sort network in an MPC setting is described in details. Section 4.3 describes how we sort integers represented by unary counters. We explore two different settings, the sorting of distinct input values (single-input values) and the classical sorting without any multiplicity constraint (multiple-value inputs). These two different settings lead to quite different efficiency results. Finally, Section 4.4 compares the two approaches of sorting networks and unary sorting and draws conclusions.

4.1 Preliminaries

4.1.1 Our Contributions

This chapter provides a secure implementation of a classical sorting network. It shows that there is no overhead linked to security when using sorting networks. This is possible thanks to the input-independent structure of sorting networks. Our implementation fits perfectly with the theoretic complexity. Therefore, we used it to benchmark our other sorting algorithms.

We detail two algorithms that sort integers expressed in a unary representation. These protocols offer a good alternative to the existing solutions. They can be used as primitives for more complex algorithms using a unary representation. It avoids extra costs of changing the representation of the integers. Our unary sorting protocols do not use comparisons to sort integers. Finally, we compare the sorting network approach and the unary representation approach. We describe when it is preferable to use one solution or the other.

4.1.2 Related Works

A lot of traditional sorting algorithms have been developed. We briefly mention some famous comparison-based sorting algorithms. In 1960, Hoare developed the Quicksort algorithm [63]. Then, the algorithm was studied thoroughly and improved by Sedgewick [64], [65]. In 1964, Williams invented the Heapsort algorithm [66] that was improved the same year by Floyd [67]. In 1973, Neumann proposed a sorting by merging algorithm [68]. All these sorting algorithms based on comparisons cannot perform better than $\mathcal{O}(n \cdot \log n)$.

Since their introduction by Batcher in 1964 [69], sorting networks have been an open research problem. Nevertheless, there has been little improvement in practical and generic sorting networks. To cite but a few, Ajtai *et al.* proposed the AKS sorting network which is impractical but achieves optimal asymptotic complexities [70]. Parberry proposed the pairwise sorting network with the same complexity as Batcher's Odd-even merge sort [71]. Networks that almost correctly sort the inputs have also been proposed as well as networks tailored for a predefined (and usually small) number of inputs [68]. We do not address these networks here because they do not fit our purpose. Instead, we use the well-known Odd-even merge sort to benchmark our protocols.

Unary counters have been used by Launchbury *et al.* to develop secure lookup tables [59] and by Toft to solve linear programs [58]. They have also been

used by Aly *et al.* to compute a shortest path in a graph securely [2]. In the present work, we use unary counters to achieve sorting protocols without any comparison.

Goodrich described a randomized and data-oblivious version of the Shellsort algorithm that achieves optimal $\mathcal{O}(n \cdot \log n)$ size and $\mathcal{O}(\log n)$ depth [72]. It succeeds in sorting any given input permutation but does not guarantee an exact sorting, it only sorts inputs with a very high probability.

In their paper, Secure Multi-Party Sorting and Applications [73], Jónsson *et al.* proposed an MPC sorting protocol based on techniques from sorting networks. They implemented and evaluated it on the Sharemind MPC platform. Their protocol is of independent interest but can also be used as a building block in the weighted set intersection problem, for example. Our Odd-even merge sort implementation relies on techniques similar to those introduced by Jónsson *et al.* Our goal is not to provide a novel sorting protocol based on sorting networks but to compare different sorting networks, to select the best one and then to provide a basis to evaluate the ranges of efficiency of our unary sorting protocols. We choose to implement them on a more generic platform: our protocols can be used for more than 3 parties while Sharemind only enables exactly 3 parties. This restriction might be annoying in an auction process, for example. Moreover, Sharemind only deals with passive security.

In the paper, Generic Constant-Round Oblivious Sorting Algorithm for MPC [74], Zhang introduced two secure sorting protocols. The first one is based on Seward's counting sort algorithm. The algorithm counts the number of occurrences for each possible value and the sorted sequence is then constructed using these counts [62]. As it does not use any comparison, the lower bound is no more $\mathcal{O}(n \cdot \log n)$ like it is for all the comparison-based sorting algorithms. If all the n inputs are in an interval of size b, the complexity is $\mathcal{O}(b \cdot n)$. The algorithm uses the constant-round bit-decomposition protocol and unbounded fan-in AND gate of Damgård *et al.* [57]. The second protocol introduced by Zhang is based on Arulanandham *et al.*'s bead sort [75], [76]. It sorts positive integers in a kind of natural way. The positive integers are represented as beads attached to rods. At the beginning, they appear as suspended in the air before sliding down attracted by gravity and sorting by themselves. Arulanandham *et al.* used a special hardware to simulate the falling of the beads while Zhang reproduced it in software using again counters. The complexity is $\mathcal{O}(b \cdot n)$ comparisons.

Hamada *et al.* proposed a practically efficient multi-party sorting protocol based on the Quicksort algorithm [77]. They overcome the issues related to the data-dependent algorithms by using an oblivious shuffle on the inputs before sorting.

This shuffle avoids privacy leakage when revealing the result of comparisons. More generally, this approach can be used to convert non-oblivious comparison sort algorithms into their secure MPC counterpart.

In a second paper, Hamada *et al.* presented an even more efficient sorting algorithm based on the radix sort algorithm [78]. The radix sort algorithm sorts fixed-length integers by iteratively applying a digit-wise stable sort from the least to the most significant digit [62]. The oblivious radix sort of Hamada *et al.* also uses a shuffle before sorting and is particularly efficient if the number of parties and the size of the underlying field are small.

Bogdanov *et al.* have recently provided an overview of the existing types of methods to perform oblivious sorting [79]. They distinguished the constructions based on sorting networks [73], on bitwise representations [74] and on oblivious shuffling [77]. They also proposed two optimization directions. The first one is to use vector operations to reduce the number of sub-protocol invocations. They described a naive comparison sort where all comparisons are done in one round but where each input is compared with every other input. They claimed that this parallelizable protocol can work faster for small input vectors than sequential protocols due to a smaller network latency. The second optimization direction is to use sorting networks with bitwise shared representations. Most of the time, the technique used for comparison is to first compute the sharing of the values' individual bits and then to execute the comparison on the bitwise representation. For multiple comparisons on the same values, as it is the case for sorting, it might be more efficient to compute once the bitwise representation and to perform all comparisons on these expressions.

Table 4.1 summarizes the complexity of the aforementioned protocols. The variable n represents the number of inputs and b the range of these inputs.

4.2 Secure Sorting using Sorting Networks

Secure sorting requires data-independent operations. Unfortunately, most sorting *algorithms* (Quicksort, Shellsort,...) exploit the outcome of previous comparisons to compute the next ones: they are adaptive or data dependent. It is not possible to use them as such to develop secure protocols. However, sorting *networks* (Bitonic sort, Odd-even merge sort,...) use a sequence of comparisons set in advance: they are oblivious or data independent. For this reason, they are good candidates for the design of secure sorting algorithms.

Sorting protocol	**Number of rounds**	**Complexity**
Randomized Shellsort	$\mathcal{O}(\log n)$	$\mathcal{O}(n \cdot \log n)$ comparisons
Odd-even merge sort	$\mathcal{O}(\log^2 n)$	$\mathcal{O}(n \cdot \log^2 n)$ comparisons
Counting Sort	$\mathcal{O}(1)$	$\mathcal{O}(b+n)$ bit-decompositions + $\mathcal{O}(b \cdot n)$ fan-in AND gates
Arrayless bead Sort	$\mathcal{O}(1)$	$\mathcal{O}(b \cdot n)$ comparisons
Quicksort (average/worst)	$\mathcal{O}(\log n)$ / $\mathcal{O}(n)$	$\mathcal{O}(n \cdot \log n)$ / $\mathcal{O}(n^2)$ comparisons
Radix Sort	$\mathcal{O}(1)$	$\mathcal{O}(n \cdot \log n)$ comparisons
Single-value unary Sort	$\mathcal{O}(b)$	$\mathcal{O}(b \cdot n)$ multiplications
Multiple-value unary Sort	$\mathcal{O}(b+n)$	$\mathcal{O}((b+n)^2)$ multiplications

Table 4.1: Comparison of the current oblivious sorting protocols.

4.2.1 Sorting Networks

Sorting networks were introduced by Batcher in 1964 [69]. They are made up of wires and comparators. Figure 4.1 shows a comparator connecting two wires and sorting their value: the minimum value comes out on the upper wire while the maximum value comes out on the lower one. A sorting network is a set of predefined comparators that will sort all possible inputs. Execution is therefore completely oblivious to the inputs.

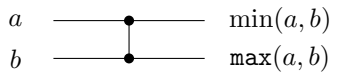


Figure 4.1: A comparator in a sorting network.

Two different efficiency metrics are used to evaluate a sorting network. The *network depth*, also called the network delay, is the number of rounds needed to sort inputs. The *network size*, also called the network cost, is the total number of comparators in the network. Table 4.2 shows three sorting networks described by Batcher [80] with their asymptotic complexities. Bubble sort is a basic sorting network with an $\mathcal{O}(n^2)$ depth and an $\mathcal{O}(n^2)$ size. Bitonic sort and Odd-even merge sort both have an $\mathcal{O}(\log^2 n)$ depth and an $\mathcal{O}(n \cdot \log^2 n)$ size. Although the asymptotic size of both networks is equal, the exact number of comparators is different: $n \cdot (\log^2 n + \log n)/4$ comparators for the Bitonic

sort and $n \cdot (\log^2 n - \log n + 4)/4 - 1$ comparators for the Odd-even merge sort. It corresponds to 28160 comparators for a 1024-input Bitonic sort and to 24063 comparators for a 1024-input Odd-even merge sort. This difference in the "constants" matters with regards to practical results.

In 1983, Ajtai *et al.* presented the AKS sorting network with an $\mathcal{O}(\log n)$ depth and an $\mathcal{O}(n \cdot \log n)$ size [70]. Although this network achieves optimal asymptotic complexities, it is unusable in practice due to its high hidden constants: it is competitive as from 2^{6000} inputs. In 1992, Parberry presented the Pairwise sorting network, the first sorting network to be competitive with the Odd-even merge sort for all input values [71]. They both have exactly the same size and depth. To the best of our knowledge, Parberry's Pairwise sorting network and Batcher's Odd-even merge sort are the most efficient networks to sort n inputs with practical values of n and $n > 16$ without any error.

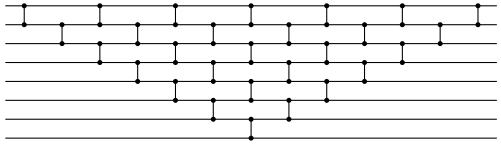
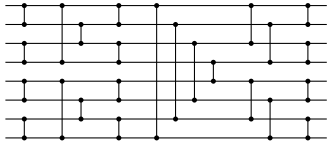
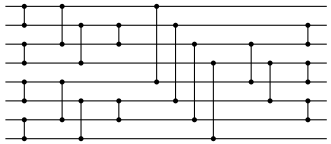
	Figure for $n = 8$	Network depth (Number of rounds)	Network size (Number of comparisons)
Bubble sort		$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Bitonic sort		$\frac{\log n \cdot (\log n+1)}{2}$ $= \mathcal{O}(\log^2 n)$	$\frac{n \cdot (\log^2 n+\log n)}{4}$ $= \mathcal{O}(n \cdot \log^2 n)$
Odd-even merge sort		$\frac{\log n \cdot (\log n+1)}{2}$ $= \mathcal{O}(\log^2 n)$	$\frac{n \cdot (\log^2 n-\log n+4)}{4} - 1$ $= \mathcal{O}(n \cdot \log^2 n)$

Table 4.2: Comparison of three well-known sorting networks.

4.2.2 Protocol and Implementation Prototype

Our protocols are built on top of the arithmetic black-box functionality $\mathcal{F}_{ABB}$ of Damgård and Nielsen [51]. It allows computing the complexity with regards to the number of multiplications and comparisons performed. We refer to Section 2.5 for more details. As usual, the notation $[x]_q$ or more simply $[x]$ represents a share of x over $\mathbb{Z}_q$ with q prime. The notation $[\mathbf{X}]$ stands for a list or a table of shares. Finally, the notation $[a < b]$ represents a share of 1 if $a < b$ and a share of 0 if $a > b$.

Sorting networks have a predefined structure and are based on a unique element (a comparator). The comparator is the only element that needs to be secured. An oblivious comparator can be achieved, among others, by the following compare-exchange function:

$$\begin{aligned} [x] &= [a < b] \\ [\min(a,b)] &= [x][a] + (1 - [x])[b] \\ [\texttt{max}(a,b)] &= [x][b] + (1 - [x])[a] \end{aligned}$$

This function uses one secure comparison and four secure multiplications. However, the multiplication of two shares is a costly primitive, compared to the addition of two shares that comes for free in MPC (no communication cost). In order to use only one comparison and one multiplication, this function can be rewritten as follows:

$$\begin{aligned} [x] &= [a < b] \\ [y] &= [x] * [a - b] \\ [\min(a,b)] &= [b] + [y] \\ [\texttt{max}(a,b)] &= [a] - [y] \end{aligned}$$

The implementation is realized thanks to the Virtual Ideal Functionality Framework (`http://viff.dk`) developed by Geisler *et al.* at Aarhus University [48]. For the comparison protocol, we use Damgård *et al.*'s protocol [57] improved by Reistad and Toft [81]. We only present our prototype of the Odd-even merge sort network (see Figure 4.2). A sort function based on the Bitonic sort is provided as a part of the VIFF framework. Although the number of rounds is equal, the Odd-even merge sort outperforms the Bitonic sort in terms of the exact number of comparisons.

The odd-even merge sort algorithm is based on a merge algorithm that merges

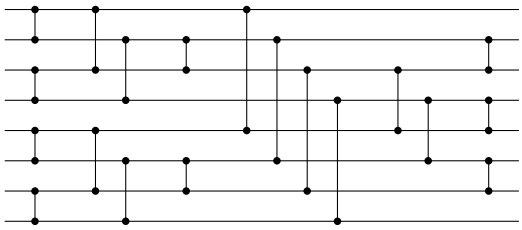


Figure 4.2: Odd-even merge sort network for 8 inputs.

two sorted halves of a sequence to a completely sorted sequence. The sorting is achieved by recursive applications of the merge algorithm.

The core function of the secure sorting based on the Odd-even merge sort is identical to a traditional implementation, except for the comparator block: it is replaced by the secure exchange function.

```
def secure_odd_even_merge_sort(self,array):

    # If x contains more than one element, split x in the middle,
    # sort recursively the upper and the lowest half
    # before finally merging them
    def odd_even_merge_sort(x):
        if len(x)<=1:
            return x
        else:
            first = odd_even_merge_sort(x[:len(x)/2])
            second = odd_even_merge_sort(x[len(x)/2:])
            return odd_even_merge(first + second)

    # Form recursively the even and odd subsequences
    def odd_even_merge(x):
        if len(x) == 2:
            x[0], x[1] = oblivious_compare(x[0],x[1])
            return x
        else:
            # even subsequence
            e = odd_even_merge(extract(0,x))
            # odd subsequence
            o = odd_even_merge(extract(1,x))
            result = interleave(e,o)
            return odd_even_compare(result)

    # Extract either the odd or the even subsequence
    # depending on the start value
    def extract(start, x):
        res = range(len(x)/2)
```

```
        ind = range(start,len(x),2)
        for i in ind: res[i/2]=x[i]
            return res

    # Insert the odd and the even subsequences
    # after the oblivious comparisons
    def interleave(x,y):
        res = range(len(x)+len(y))
        for i in range(len(x)):
            res[2*i]=x[i]; res[2*i+1] = y[i]
            return res

    # Perform the last comparison step
    def odd_even_compare(x):
        for i in range(1,len(x)-1,2):
            x[i], x[i+1] = oblivious_compare(x[i],x[i+1])
            return x

    # Perform the oblivious exchange (compare-swap)
    def oblivious_compare(i, j):
        b = i <= j
        b_ai_aj = b * (i - j)
        return j + b_ai_aj, i - b_ai_aj

    newarray = odd_even_merge_sort(array)
    return newarray
```

Listing 4.1: Odd-even merge sort

Figure 4.3 shows in grey the experimental running time of this Odd-even merge sort prototype. The black curve corresponds to the theoretic number of comparators multiplied by their execution time (0.12 second for one comparator). Times correspond to computations between three entities (threshold = 1) on a single machine. Figure 4.3 shows that our practical results perfectly fit the expected complexity curve.

In this scenario, there is no extra cost due to the passage to the secure solution. The asymptotic complexities are the same as the one described in Table 4.2. The extra costs only come from the higher cost of the basic secure operations like multiplication.

Except for some computations, the secret sharing for example, comparators are the only operation to take part to our protocol. Our comparator is composed of one comparison and one multiplication. The secure comparison protocol for comparing secret variables of l-bit size uses $\mathcal{O}(l)$ secure multiplications. With $l = 32$ and passive security, one comparison is achieved with 165 multiplications. It results in our secure Odd-even merge sort using $165 \cdot (\frac{n \cdot (\log^2 n - \log n + 4)}{4} - 1)$ secure multiplications. Therefore, we have a strong interest in finding a sorting

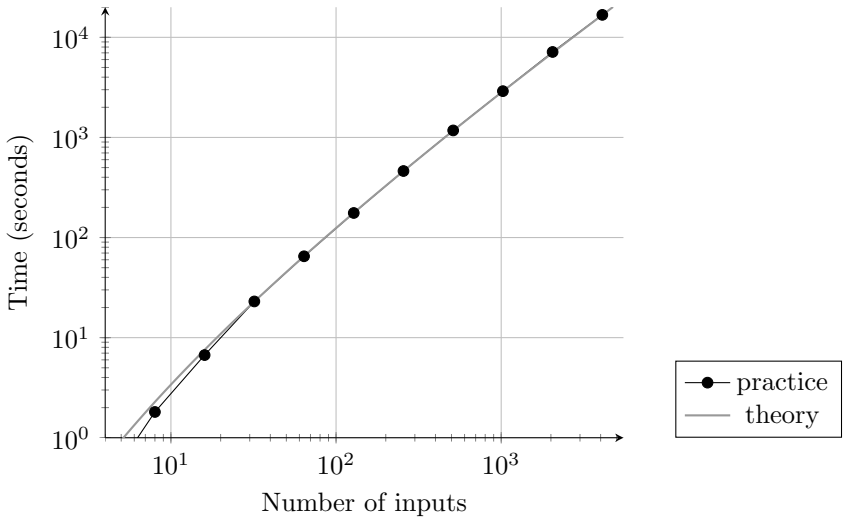


Figure 4.3: Running time of our secure Odd-even merge sort prototype.

method, maybe less efficient in a traditional setting, which avoids the costly comparisons.

4.3 Secure Sorting using Unary Counters

We propose two protocols using unary counters. Both of them sort inputs in a given interval. The first one only sorts different inputs (unary sort with single-value inputs) while the second one admits inputs multiplicity (unary sort with multiple-value inputs).

4.3.1 Unary Sorting with Single-Value Inputs

Let us show an example of the prototype where we chose $n = 4$ and $b = 5$. The list ([2] [0] [1] [4]) becomes, in a unary representation, the array:

$$[\mathbf{A}] = \begin{pmatrix} [0] & [0] & [1] & [0] & [0] \\ [1] & [0] & [0] & [0] & [0] \\ [0] & [1] & [0] & [0] & [0] \\ [0] & [0] & [0] & [0] & [1] \end{pmatrix}$$

The algorithm sums each column of the array $[\mathbf{A}]$ in `sum_vector`, that is

`sum_vector` = ([1] [1] [1] [0] [1]). For each position i in `sum_vector`, a share of 1 indicates that the value i is in the list, while a share of 0 indicates that the value i is not in the list. The `sorted_vector` will contain the sorted list after the execution of the algorithm. It is initialized with shares of 0. The `index` vector is initialized with shares of 0 except `index`(0) that is initialized with a share of 1. This vector maintains the current position in the `sorted_vector`, that is the position where the next sorted element has to be obliviously inserted.

To obtain the sorted list, we need to obliviously extract the positions that contain a share of 1 and eliminate the positions that contain a share of 0. It is achieved using the loop at Line 15. The loop goes through all the elements i in the interval (of size b). At each iteration, it inserts a share of i in a temporary vector at the index given by the `index` vector. If `sum_vector`(i) = [1], it obliviously updates the `sorted_vector` and obliviously increments the index. Conversely, if `sum_vector`(i) = [0], it does not change neither the `sorted_vector` nor the index but every entry is updated with a different share of the same value to prevent the leakage of information. In our example, the algorithm outputs the sorted list [**A**] = ([0] [1] [2] [4]).

```
1  def unary_sort_single_value(self, matrix):
2
3      # Number of elements to sort
4      n = matrix.shape[0]
5      # Size of the interval (size of the unary representation)
6      b = matrix.shape[1]
7      # Vector where the i-th entry is the sum of the i-th column of
           the input matrix.
8      sum_vector = matrix.sum(axis=0)
9      # Initialize the vector that will contain the sorted list with
           shares of 0 (self.zero is not described here).
10     sorted_vector = [self.zero]*n
11     # Initialize the index vector that will maintain the current
           position in the sorted_vector (where the next sorted
           element has to be inserted).
12     index = [self.zero]*n
13     index[0]=self.one
14
15     for i in range(b):
16         # The condition is 1 if there is actually an element at
               the i-th entry and is 0 otherwise.
17         condition = sum_vector[i]
18         # Insert a share of i at the index entry. n mult.
19         temp_vector = self.update_vector_private(sorted_vector,
               index, i)
20         # ''Remove'' obliviously the just inserted element if the
               condition is 0 (concretely, if the condition is 0, the
               update in temp_vector is not taken into account). n
```

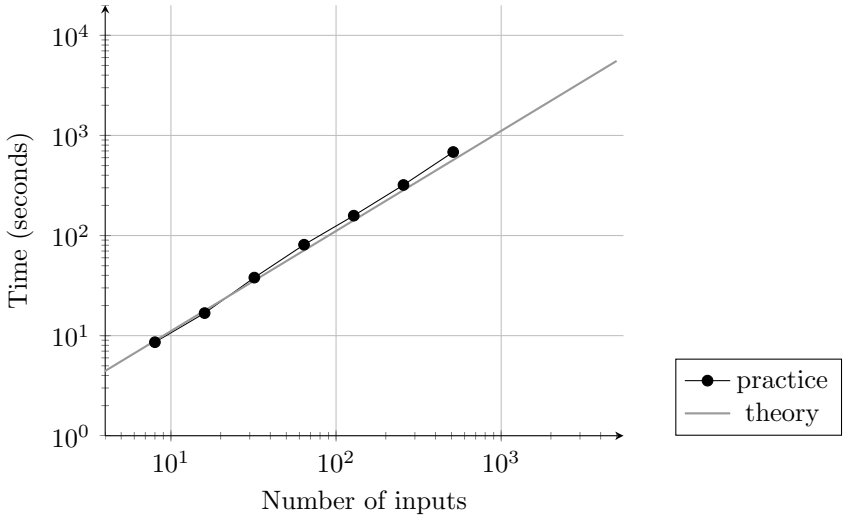


Figure 4.4: Running time of our unary sort prototype with b = 512.

```
            mult.
21          sorted_vector = (temp_vector-sorted_vector)*condition+
                sorted_vector
22          # Increment the position if the (private) condition is 1.
                n mult.
23          index = self.update_index(index, condition)
24
25      return sorted_vector
```

Listing 4.2: Unary sort for single-value inputs

Lines 19, 21 and 23 perform each n multiplications and the loop is executed b times (with n the number of elements to sort and b the size of the interval). The prototype requires $3.b.n$ multiplications.

Figures 4.4 and 4.5 show the practical results we get compared to the expected theoretic complexity.

4.3.2 Unary Sorting with Multiple-Value Inputs

This second algorithm is based on the previous one. We use the same variables and introduce an extra public `values` vector. It contains the values 0,1,2,...,b-1. The next values to insert in the `sorted_vector` are picked among them. We

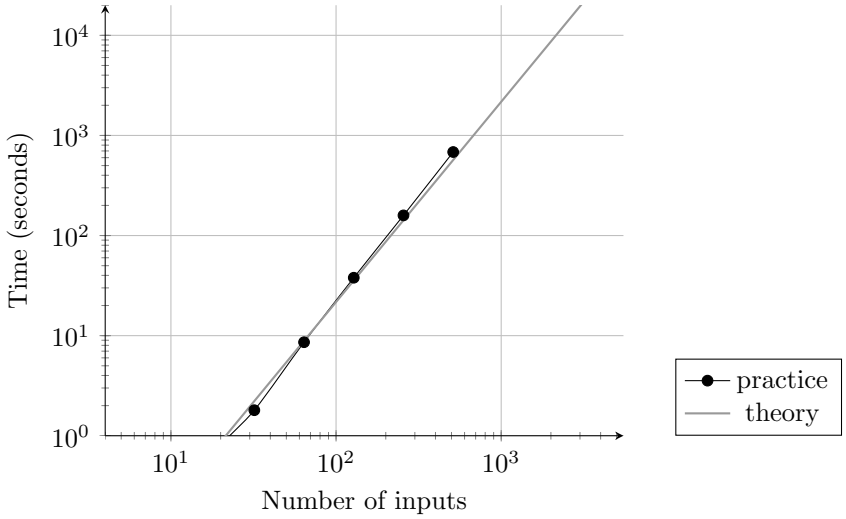


Figure 4.5: Running time of our unary sort prototype with b = n.

also introduce the private vector `values_index`. The vector maintains the current position (index) in the values vector.

The sorted list is recursively constructed by the loop at Line 20 of the following algorithm. If `current_multiplicity` $\geq$ [1], it obliviously updates (decrements) the `sum_vector`, updates the `sorted_vector` and increments the `sorted_vector_index`. However, it does not increment the `values_index` (the non-zero multiplicity indicates that might have to insert multiple times the same value). Conversely, if `current_multiplicity` = [0], it updates the `values_index`.

```
1  def unary_sort_multiple_value(self, matrix):
2
3      # Number of elements to sort
4      n = matrix.shape[0]
5      # Size of the interval (size of the unary representation)
6      b = matrix.shape[1]
7      # Vector where the i-th entry is the sum of the i-th column of
           the input matrix.
8      sum_vector = matrix.sum(axis=0)
9      # Initialize the sorted_vector with shares of 0 (self.zero and
           self.one not described here). This vector will later
          contain the sorted list.
10     sorted_vector = [self.zero]*n
11     # Initialize the sorted_vector_index vector with shares of 0
          at all entries except the first entry. This vector maintain
           the current position in the sorted_vector (the index where
```

```
        the next sorted element has to be inserted).
12      sorted_vector_index = [self.zero]*n
13      sorted_vector_index[0]=self.one
14      # Initialize the values vector with the values 0,1,2,...,b-1.
            The next element to insert in the sorted_vector is picked
            in this list.
15      values = arange(0,b,dtype=object_)
16      # Initialize the values_index vector with shares of 0 except
            for the first entry. The vector maintains the current
            position (index) in the values vector.
17      values_index = [self.zero]*b
18      values_index[0] = self.one
19
20      for i in range(n+b):
21          # Values_index is private but the vector values is public.
                Multiplications are ''free''.
22          current_value = dot(values,values_index)
23          # b mult
24          current_multiplicity = dot(sum_vector,values_index)
25          # Set the condition to 0 if the current multiplicity is 0.
                Set the condition to 1 otherwise (it means there is at
                least one occurence of the current value to put in the
                sorted list). n mult.
26          condition = 1-self.polynQ(current_multiplicity,n)
27
28          # Update sum_vector if the condition is 1. 2*b mult.
29          temp_vector_1 = self.update_vector_private(sum_vector,
                values_index, current_multiplicity-1)
30          sum_vector = (temp_vector_1-sum_vector)*condition+
                sum_vector
31
32          # Add current_value in sorted_vector if the condition is
                1. 2*n mult.
33          temp_vector_2 = self.update_vector_private(sorted_vector,
                sorted_vector_index, current_value)
34          sorted_vector = (temp_vector_2-sorted_vector)*condition+
                sorted_vector
35
36          # Increment the sorted_vector_index if the condition is 1.
                n mult.
37          sorted_vector_index = self.update_index(
                sorted_vector_index, condition)
38          # Increment the values_index if condition is 0. b mult.
39          values_index = self.update_index(values_index, 1-condition
                )
40
41      return sorted_vector
```

Listing 4.3: Unary sort for multiple-value inputs

The function polynQ evaluates the function $P(x)$ which returns 1 if $x = 0$ and 0 if $0 < x <$ multmax.

$$P(x) = \frac{(x-1)\cdot(x-2)\cdot(x-3)\cdot \quad \cdots \quad (x-\text{multmax})}{(-1)\cdot(-2)\cdot(-3)\cdot \quad \cdots \quad (-\text{multmax})}$$

```
# The function returns 1 if x = 0 and 0 otherwise. multmax mult.
def polynQ(self,x,multmax):
    result = 1
    div = 1
    for i in range(1,multmax+1):
        result = result*(x-i)
        div = div*(-i)
    diva = self.Zp(div)
    invdiv = invert(diva)
    return (result*invdiv)
```

Lines 22 , 27, 28 and 37 perform b secure multiplications while Lines 24, 31, 32 and 35 perform n secure multiplications. The loop is executed $n + b$ times. The prototype requires $4 \cdot (n + b)^2$ multiplications.

4.3.3 Discussion

As seen in previous sections, theory and practice fit quite well. Therefore, we compare the theoretic complexities. Figure 4.6 shows the ranges of usability of our unary sorting for single-value inputs. The unary sort with single-input values of an interval size of $b = n$ is very interesting to determine these ranges because it gives the best results we can hope with the unary approach. We can see that it is useless to use a unary sorting with more than 1500 values. In that case, the odd-even merge sort always outperforms the unary sort. With less than 1500 inputs, the ranges of interest depend on the size of the interval b. For example, with a size $b = 256$, a unary sort approach is worthy with input size $20 \leq b \leq 1500$. For less than 20 inputs, it becomes too onerous to compute a unary representation of size 256 compared to the few computations needed by a sorting network. However, if the inputs are in a smaller interval, for example, $b = 64$, it is still interesting. If we consider the memory usage and work on GF(256), this approach is worth interest even in different gaps.

Figure 4.7 shows the ranges of usability of our unary sorting for multiple-value inputs. The unary sort with multiple-value inputs is only competitive for very small intervals of size 32 or 64. However, once again if memory is the most important constraint, the multiple-value inputs protocol is interesting even for larger values.

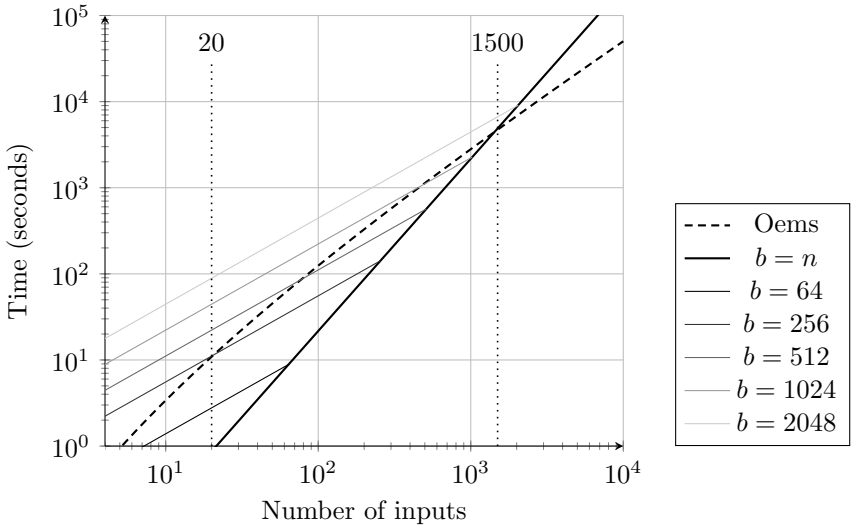


Figure 4.6: Benchmark of Unary sort with single-value inputs and Oems (holds for a comparison costing 165 multiplications).

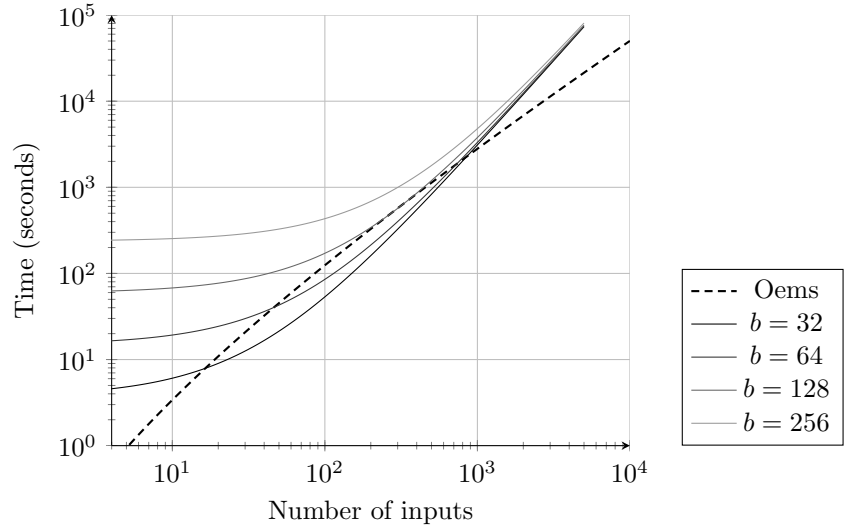


Figure 4.7: Benchmark of Unary sort with multiple-value inputs and Oems (holds for a comparison costing 165 multiplications).

4.4 Conclusion

When designing secure algorithms, the first obstacle is about the different efficiency metrics: the traditional complexity metrics do not transpose to secure computations. For example, it is more expensive, by more than two orders of magnitude, to compare values than to multiply them. This means that the most efficient traditional algorithms might not be the most efficient once adapted, when possible, to the secure setting. The second main difficulty is the leakage by data structure. Most algorithms have an execution flow that depends on the data that are manipulated. Control flow happens as a function of data that must be kept secret (the result of branching or loop exit conditions, for instance).

A secure sorting based on sorting networks is a good choice if we know little information about the inputs, for example, if we do not know the gap between the maximum and minimum values or if we do not know if there are or not repeated inputs. This sorting is also interesting if we need to sort without any error. It can be used easily as a sub-protocol. The unary sort is a good alternative if the numbers to sort are concentrated in a very close gap. It can be quite interesting if we know the distribution of the inputs in advance and if we can exploit this information. The unary sort offers a practical solution for small input size given the particular computational metric. Moreover, we can use it on the field GF(256) due to the fact that we only compute on bits. The unary sort can therefore be run on devices with not much memory.

Chapter 5

Securely Solving a Fair Division Problem

Fair division is a well studied problem in game theory. The fair division problem of cake-cutting consists in dividing a heterogeneous good so that all parties believe they have received a fair share. The problem is interesting when parties value the pieces of the heterogeneous good differently. This potentially allows the k parties to receive more than $1/k$ of the value of the good according to their own preferences. Beyond the metaphor of cutting a cake, fair divisions are advised in a wide variety of situations: to divide goods in an inheritance, to distribute chores or to split the costs of building a shared road among the users.

The goal of this chapter is to present secure solutions to the cake-cutting problem. These cryptographic solutions address important shortcomings of traditional game-theoretic procedures. Traditional procedures disclose some of the parties' preferences and offer other parties the possibility to dynamically adjust their behaviour in order to obtain unfair advantages.

Section 5.1 describes our contributions, the related works in game theory and the links between cryptography and game theory. Section 5.2 describes the modelling of the cake-cutting problem, that is the modelling of the parties' preferences as well as the function that gives an equitable division for two and for k parties in the presence of a mediator. This section is a main contribution to the chapter. It shows how to use the cryptographic primitives to solve the fair division problem. Section 5.3 details the secure cake-cutting protocol and gives the implementation results. Finally, Section 5.4 concludes and gives some open problems.

5.1 Preliminaries

Two different properties are quite useful in division problems: fairness and efficiency. A division is *efficient* if no other division is strictly better for at least one party and as good for the others. The importance of efficiency is obvious but the perception of fairness is also very important. For example, to divide an inheritance between siblings, it is important that each sibling does not envy the share another one has got. Fairness seems more important than efficiency in this context [82].

Different criteria of fairness can be used to divide a cake. A *proportional* division guarantees each of the k parties to receive at least $1/k$ of the cake according to his valuation. A division is *envy-free* if the resource is divided in such a way that no one will prefer another party's share. If all the parties get the same proportion, according to their valuation, the division is called *equitable*. Illustrations and detailed definitions of these criteria are widely available in the literature [83], [84], [85].

Proportionality and envy-freeness are equivalent with only two parties. With more than two parties, a proportional sharing does not guarantee that the parties will not be envious. However, an envy-free division always gives the parties a proportional share. Equitability seems to be a strong criterion of fairness. However, it must be used together with another criterion: an equitable division can even fail to be proportional. Equitability implies neither envy-freeness nor proportionality and none of these properties implies equitability.

5.1.1 Our Contributions

In this chapter, we give a secure multi-party protocol for an equitable cake-cutting. Contrary to the game-theoretic procedures, our solution preserves the privacy of the preferences. The parties only learn the final cutting point that is strictly minimal. Moreover, our solution leads to the high payoffs of a mediated division by emulating the mediator. Appendix B gives more details about the expected payoffs of mediated games. Parts of this work were written in collaboration with C. Petit and O. Pereira and presented at WISSec 2010 [1].

Our protocol is more efficient than generic techniques and, therefore, usable in practice. There is no known procedure in game theory to achieve, without a physical mediator, an equitable division of a heterogeneous cake with uncut pieces. This new result is possible by using MPC to emulate a mediator.

We introduce step functions to model the parties' preferences because they are

well appropriate to the restrictions inherent to the secure multi-party protocols. Furthermore, step functions appear to arbitrarily closely approximate any reasonable utility function. This class of functions enables us to write an equitable cake allocation procedure for MPC.

Then, we use the cake allocation procedure in a secure multi-party protocol. The cake-cutting protocol relies on a secure protocol for comparison [57] and on a secure protocol for inversion of non integer values [86]. The properties of the cake-cutting protocol depend on the properties of these building blocks.

In this chapter, we distinguish the 2-party and the k-party cases. We show the mathematical model in both cases but only detail the secure protocol in the 2-party case. The k-party secure protocol can easily be deduced from the mathematical model. There are two major differences for the k-party case. The solutions are multiple: to each ordering of the players corresponds an equitable division. The equitability does no longer guarantee proportionality.

5.1.2 Related Works

Many applications of MPC have been developed in the last years. Related works about these applications can be found in Section 2.4.1. This section briefly summarizes related works in game theory.

Game theory provides solutions to fair divisions in a lot of interesting cases. The basic one with two players is "I cut, you choose". Alice cuts the cake into two parts she thinks equal and Bob chooses the part with the greatest utility according to his preferences. This procedure is not equitable and, like all the game-theoretic procedures, suffers from a lack of secrecy: Bob gets a lot of information about Alice's preferences, since Alice cuts the cake into two halves she considers exactly equal. However, Alice does not know how far Bob prefers the share he has chosen to the other one. Appendix C describes this game in more details.

More complex game-theoretic solutions exist for an infinitely divisible cake as well as for indivisible goods [82]. There are two kinds of procedures: discrete and continuous moving-knife procedures. Brams and Taylor give an unbounded discrete procedure for an envy-free division between k players [87]. Brams, Taylor and Zwicker describe a moving-knife procedure for an envy-free division between four players [88]. Like "I cut, you choose", these procedures reveal a lot of information about the players' preferences but this disclosure of information is even more annoying, as it enables the players to adapt their strategy in order to obtain a bigger piece of cake.

The presence of a mediator improves and simplifies fair divisions. The players can privately communicate their preferences to him and he can then suggest a fair division satisfying all parties. A mediated division reveals nothing about the players' preferences except for the information that can be deduced from the cutting points. Furthermore, such a division potentially leads to much higher payoffs, that is, the players receive a bigger piece of cake according to their own valuation. A trusted mediator is, however, not very easy to find and his existence is unsafe. A person who knows the preferences of each party will focus all attacks on himself.

Dodis *et al.* also use cryptographic protocols to solve extensive form games [89]. Appendix B provides a short outline on the standard definitions of game theory. It shows how a game-theoretic mediator can improve the expected payoffs of the players. However, the game-theoretic mediator is limited by the rationality of the parties. Appendix C illustrates these definitions on a cake-cutting problem (I cut, you choose).

5.1.3 Bridging Cryptogaphy and Game Theory

Bridging cryptography and game theory seems very interesting but challenging because of their different settings. This research direction was initiated by Dodis and Rabin [90] and Izmalkov *et al.* [91]. Table 5.1 summarizes the key differences between the cryptographic and game-theoretic settings. Katz provides more details in his report [92].

	Cryptography	**Game theory**
Players/Parties	totally honest or malicious	always rational
Incentive	outside the model	payoff
Solution drivers	secure protocol	equilibrium
Privacy	goal	means
Trusted party	in the ideal model	in the actual game

Table 5.1: Cryptography and Game Theory Settings.

In cryptography, some players are supposed to follow the protocol honestly while other malicious players can deviate from it. In the honest-but-curious model, all players, even the corrupted ones, follow the protocol blindly without any incentives. In the malicious model, however, the corrupted players may behave in a completely unexpected and even irrational manner. In game theory, all players have a rational behaviour: they all follow the protocol only if it is

in their own and best interest in terms of payoffs. Rationality is a common knowledge, i.e., players know that all players are rational and use this knowledge to make their strategic decisions. These different settings are quite difficult to compare.

In the cryptographic settings, the objective is to achieve a secure protocol designed to eliminate the trusted party. The functionality of the mediator is shared among the players and he thus only exists in the ideal model. The goal of the secure multi-party protocol is to assure the correctness of the outputs and the privacy of the inputs even when some parties are cheating. In the game-theoretic settings, the goal is to reach an equilibrium, i.e., a situation where no player has an incentive to deviate from his strategy. Privacy is a central issue in cryptography while it is only a means in game theory.

A secure multi-party protocol leaks no information during its execution. The players can thus not adapt their preferences and the protocol leads to a more equitable division. We work under the settings of a continuous cake. The cutting points have therefore non integer values. This is a difficulty because MPC deals with discrete values.

5.2 Modelling the Cake-Cutting Problem

To achieve a mediated equitable cake division, the players have to express their preferences in a digital way and to agree on a function that, given their preferences as input, computes the cutting points as output. Each player explicitly tells how much he values the different parts of the cake through a utility function defined on the "cake interval". We represent the cake by a given interval of $\mathbb{R}$. Utility functions cannot be general to get a simple analytic expression of the cutting points.

We express the players' preferences for the different pieces of cake by a step function. This is a convenient choice: it is very easy to calculate the integral of a step function and to approximate more general functions by step functions. To get a simple analytic expression, we add other assumptions. All the steps of the function should have the same width and an integer value. The cake should be valued on $[0, N]$ with N equal to the number of steps of the step function (the steps have a width of 1). Adapting the size of the cake to the number of steps of the function makes it possible to deal with integer values for the integration: the integral of a utility function on a unitary interval is an integer value. This simplifies the integration and since secure comparison is possible, it is not a problem to proceed by parts.

No one is favoured: all players have the same total utility for the cake. Let $U \in \mathbb{R}^+$ be the utility of a player for the whole cake. Moreover, all the utilities for each interval are supposed to be strictly positive integers.

5.2.1 Equitable Division for 2 Players

In this section, we focus on determining the functions that two players have to securely evaluate in order to find the cutting point. The inputs of a player are the values of his N-step function on each interval. The output corresponds to the cutting point. Equitability is required for the sharing. Every player gets exactly the same proportion of the cake according to his own valuation.

Let α be the cutting point. The utility function of the first player (P_1) is defined by $f_1(x) = u_1$ on $[0, 1]$, $f_1(x) = u_2$ on $]1, 2]$, ... , $f_1(x) = u_N$ on $]N-1, N]$. In the same way, the utility function of the second player (P_2) is defined by $f_2(x) = v_1$ on $[0, 1]$, $f_2(x) = v_2$ on $]1, 2]$, ... , $f_2(x) = v_N$ on $]N-1, N]$.

Equitability implies that

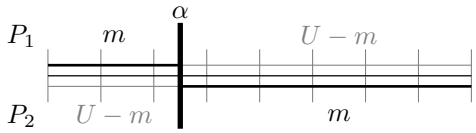
$$\int_0^{\alpha} f_1(x)\,dx = \int_{\alpha}^{N} f_2(x)\,dx.$$

The left piece of cake is arbitrarily allocated to P_1 and vice-versa for P_2. This is not a problem since the division is equitable and there are only two players. Either both players are happy with their share, or they both receive less than a half of the cake (according to their own valuation) and they envy each other. In this case, they exchange their parts. The equitable division for two players is therefore also proportional and envy-free. The following propositions show the existence and uniqueness of the cutting point for two players.

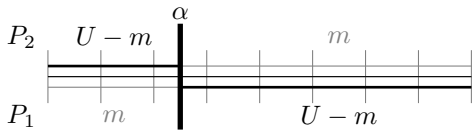
Proposition 1. *The cutting point is unique for two players.*

Proof. There are two different orderings of the two players. Let us take the ordering P_1 - P_2. Define the functions $g_1(\alpha) := \int_0^{\alpha} f_1(x)\,dx$ and $g_2(\alpha) := \int_{\alpha}^{N} f_2(x)\,dx$. Since f_1 and f_2 are bounded, g_1 and g_2 must be continuous. Now, since $g_1(0) = 0$ and $g_1(N) > 0$ while $g_2(0) > 0$ and $g_2(N) = 0$, there must be a value $\alpha \in [0, N]$ such that $g_1(\alpha) = g_2(\alpha)$. Moreover, since f_1 and f_2 are strictly positive functions, g_1 and g_2 must be strictly monotonic, and the point α satisfying $g_1(x) = g_2(x)$ must be unique.

Since $g_1(N) = g_2(0)$, the cutting point α also gives an equitable division for the ordering P_2 - P_1 (with a utility $U - m$ for each player) and since we know that



(a) Ordering P_1 - P_2



(b) Ordering P_2 - P_1

Figure 5.1: Equitable divisions for the two different players' orderings.

the equitable division is unique, we also find the equitable division corresponding to the ordering P_2 - P_1. Figure 5.1 provides an illustration of the proof. □

Proposition 2. *An equitable, envy-free and proportional division always exists for two players.*

Proof. There are two orderings of the players leading to two different divisions. One division gives a utility of m and the other of $U - m$. At least one division leads therefore to a utility of at least $U/2$, giving a proportional and envy-free division. □

Where is the cutting point? The players have to securely evaluate the following equation for $\alpha \in]i-1, i]$:

$$u_1 + ... + u_{i-1} + (\alpha_i - (i-1))u_i = (i - \alpha_i)v_i + v_{i+1} + ... + v_N \tag{5.1}$$

Equation 5.1 checks whether $\alpha \in]i-1, i]$ with $i \in \mathbb{Z}$ and $1 \leq i \leq N$. If $\alpha_i \in]i-1, i]$, it is the exact cutting point ($\alpha = \alpha_i$). The solution of the equation is given by the following equality:

$$\alpha_i = \frac{-(u_1 + ... + u_{i-1} - (i-1)u_i) + (i\ v_i + v_{i+1} + ... + v_N)}{(u_i + v_i)}.$$

In concrete terms, the function the players have to evaluate securely for $]i-1, i]$

is

$$F_i(u_1, \ldots, u_N, v_1, \ldots, v_N) = b_i \; a_i^{-1}, \text{ with}$$

$$a_i = u_i + v_i,$$
$$b_i = -\sum_{m=1}^{i} u_m + \sum_{m=i+1}^{N} v_m + i \; (u_i + v_i).$$

Binary search through the intervals. The function $F_i(u_1, \ldots, u_N, v_1, \ldots, v_N)$ is computed for each of the N intervals until the cutting point is found. In the worst case, N equations must be solved. We are going to partially solve this efficiency issue. If $\alpha_i \notin]i-1, i]$, it is not the exact cutting point but an estimation of α. It is all the more precise that $]i-1, i]$ is close to α. Thus $\alpha_i > i$ implies $\alpha > i$ and $\alpha_i < i-1$ implies $\alpha < i-1$. This makes it possible to reduce the number of iterations through a binary search algorithm. In the worst case $\log_2 N$ equations are solved. Let us show that $\alpha_i > i$ implies $\alpha > i$.

Proof. Suppose that $\alpha_i > i$ ($\alpha_i = i + \epsilon$ with $\epsilon > 0$), then the following equalities result from Equation 5.1:

$$\begin{aligned} u_1 + ... + u_{i-1} + ((i+\epsilon) - (i-1))u_i &= (i - (i+\epsilon))v_i + v_{i+1} + ... + v_N \\ u_1 + ... + u_{i-1} + (\epsilon + 1)u_i &= (-\epsilon)v_i + v_{i+1} + ... + v_N \\ \underbrace{u_1 + ... + u_i}_{\int_0^i f_1(x)dx} + \epsilon(u_i + v_i) &= \underbrace{v_{i+1} + ... + v_N}_{\int_i^N f_2(x)dx}. \end{aligned}$$

The last equality implies that

$$\int_0^i f_1(x)dx < \int_i^N f_2(x)dx,$$

which directly implies that $\alpha > i$ since $\int_0^\alpha f_1(x)\, dx = \int_\alpha^N f_2(x)\, dx$ with $f_1(x) > 0$ and $f_2(x) > 0$, for all x. □

With the binary search algorithm, the computation efficiency is improved by a factor of $\log_2 N$. However, contrary to the exhaustive search that enables checking all intervals in parallel, the binary search requires doing the computation sequentially.

An alternative solution to perform the binary search is to compare the sums of utilities $\sum_{k=1}^{i} u_k$ and $\sum_{k=i}^{N} v_k$ instead of evaluating Equation 5.1. In this

case, we share the cumulative utilities $u'_i = \sum_{k=1}^{i} u_k$ and $v'_i = \sum_{k=i}^{N} v_k$ at each iteration instead of sharing all separated utilities. It means we only share $\mathcal{O}(\log N)$ values instead of $\mathcal{O}(N)$. It does not matter if we use the solution as a subroutine (the utilities are already shared) but it is interesting otherwise.

5.2.2 Equitable Division for k Players

With three or more players, the equitability criterion is no longer sufficient to guarantee an envy-free or even proportional cutting. Indeed, it is not always possible to get such a division by letting the players simply exchange their respective share. Further, the more the number of players increases, the more difficult it is to solve a system for each combination of all cutting points in all intervals. As illustrated in Figure 5.2, there are $k-1$ cutting points $(\alpha, \beta, \gamma, \ldots)$.

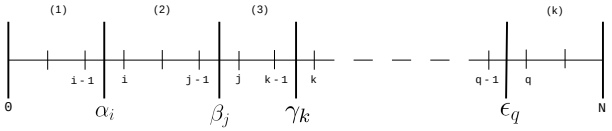


Figure 5.2: The $k-1$ cutting points.

The linear equation for two players (Equation 5.1) can be generalized by a linear system of equations for three or more players (System 5.2). This system expresses that the value the player P_1 gives to the first part of the cake has to be equal to the value the player P_2 gives to the second part, that this value has to be equal to the value the player P_3 gives to the third part and so on. Other systems are possible to achieve an equitable division. This choice was made to obtain a system with a tridiagonal matrix that is easily factorized.

Where are the cutting points? The players have to securely evaluate the following system for $\alpha \in]i-1, i]$, $\beta \in]j-1, j]$, …, $\epsilon \in]q-1, q]$:

$$\begin{bmatrix} u_i + v_i & -vj & 0 & \cdots & 0 \\ -v_i & v_j + w_j & -w_k & \ddots & \vdots \\ 0 & -w_j & w_k + x_k & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -y_q \\ 0 & \cdots & 0 & -y_p & y_q + z_q \end{bmatrix} \begin{bmatrix} \alpha_i \\ \beta_j \\ \chi_k \\ \vdots \\ \epsilon_q \end{bmatrix} = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ \vdots \\ C_{k-1} \end{bmatrix} \tag{5.2}$$

$$
\begin{aligned}
\text{with} \quad C_1 &= -\textstyle\sum_{m=1}^{i} u_m + \sum_{m=i+1}^{j} v_m + i\,(u_i + v_i) - j\,v_j, \\
C_2 &= -\textstyle\sum_{m=i+1}^{j} v_m + \sum_{m=j+1}^{k} w_m - i\,v_i + j\,(v_j + w_j) - k\,w_k, \\
C_3 &= -\textstyle\sum_{m=j+1}^{k} w_m + \sum_{m=k+1}^{l} x_m - j\,w_j + k\,(w_k + x_k) - l\,w_l, \\
&\vdots \\
C_{k-1} &= -\textstyle\sum_{m=p+1}^{q} z_m + \sum_{m=q+1}^{N} y_m - p\,y_p + q\,(y_q + z_q).
\end{aligned}
$$

The system is solved for $\alpha, \beta, \ldots, \epsilon$ in each of the N intervals (with $\alpha < \beta < \ldots < \epsilon$). In the worst case, the exhaustive search requires solving

$$\Gamma_N^{k-1} = \binom{N+k-2}{k-1} = \frac{(N+k-2)!}{(k-1)!(N-1)!}$$

systems. It means that a system is solved for each random selection with repetition and without order of $k-1$ intervals among N.

Properties of the matrix. The matrix of System 5.2 is tridiagonal with strictly positive elements on its main diagonal and strictly negative elements on the secondary diagonals (individual utilities are strictly positive). Moreover, the matrix is column diagonally dominant: ($|a_{ii}| = \sum_{i \neq j} |a_{ij}|$ for $2 \leq i \leq k-2$ and $|a_{ii}| > \sum_{i \neq j} |a_{ij}|$ for $i = 1$ and $i = k-1$). The following properties hold:

1. If a matrix A is column diagonally dominant, A admits a LU-factorization without pivoting.
2. If a tridiagonal matrix

$$A = \begin{bmatrix} d_1 & e_1 & & & \\ c_2 & d_2 & e_2 & & \\ & c_3 & d_3 & \ddots & \\ & & \ddots & \ddots & e_{n-1} \\ & & & c_n & d_n \end{bmatrix}$$

admits a LU factorization without pivoting, the two factors can be written

$$L = \begin{bmatrix} 1 & & & & \\ c'_2 & 1 & & & \\ & c'_3 & 1 & & \\ & & \ddots & \ddots & \\ & & & c'_n & 1 \end{bmatrix}, \quad U = \begin{bmatrix} d'_1 & e_1 & & & \\ & d'_2 & e_2 & & \\ & & d'_3 & \ddots & \\ & & & \ddots & e_{n-1} \\ & & & & d'_n \end{bmatrix}$$

$$\text{with} \quad d'_1 = d_1 \quad \text{and} \quad \begin{cases} c'_i &= c_i / d'_{i-1} \\ d'_i &= d_i - c'_i e_{i-1} \end{cases} \quad \text{for } i = 2 : n$$

The LU decomposition was introduced by Alan Turing in 1948 [93]. A detailed analysis can be found in standard references [94],[95].

Using the LU decomposition, it is possible to simplify the system: the matrix A is factorized into a lower triangular matrix (L) and an upper triangular matrix (U). Instead of solving $Ax = b$, we solve $Ly = b$ and $Ux = y$. Players only solve $(k-1)$ linear equations to find β and then $(k-1)$ linear equations to find α. This reduces the complexity of the protocol and is possible thanks to the special form of the matrices L and U.

The matrix A of the initial system is invertible. This can be deduced from the determinant of the matrices L and U. The determinant of L is equal to 1 and the determinant of U is equal to $d'_1 \cdot d'_2 \cdot \ldots \cdot d'_n > 0$. Let us show that the determinant of U is strictly positive.

Proof. The matrix of the initial system is tridiagonal with strictly positive elements on its main diagonal and strictly negative elements on the secondary diagonals thus $d_i > 0$ for $i = 1 : n$, $c_i < 0$ for $i = 2 : n$ and $e_i < 0$ for $i = 1 : n-1$.

$$d'_1 = d_1 > 0 \tag{5.3}$$

$$d'_i = \frac{d_i d'_{i-1} - c_i e_{i-1}}{d'_{i-1}} \tag{5.4}$$

Equation 5.4 shows that if $d'_{i-1} > 0$, then $d'_i > 0$. By recurrence, we have $d'_i > 0$ for $i = 1 : n$ and thus $d'_1 \cdot d'_2 \cdot \ldots \cdot d'_n > 0$. □

The determinants of L and U are both strictly positive, which directly implies that the determinant of A is strictly positive, that the matrix is invertible and

finally that the system has a unique solution.

5.2.3 Practical Application

In this section we give an example for which the used definition of cake-cutting makes sense. Suppose the case of two doctors who have to share a day on duty. Each of them gives his preferences for each hour (let us say between 6 am and 9 pm). Since this situation is likely to be repeated, they wish to keep their preferences private.

As each doctor wants to work as little as possible, this situation is actually the inverse of the classical cake-cutting problem. Each doctor distributes a total utility of 100 over the 15 intervals (hours) and gives a small utility for an interval he is interested in and a large utility for one he is not interested in. At the end each doctor takes the shift with the smallest utility. The work shift has to be continuous. In Section 5.3.3, we give some efficiency results.

	6 am	7 am	8 am	9 am	10 am	11 am	12 am
Doctor 1	15	7	4	1	1	1	15
Doctor 2	20	10	10	7	7	4	4

	1 pm	2 pm	3 pm	4 pm	5 pm	6 pm	7 pm	8 pm
Doctor 1	4	4	4	7	7	10	10	10
Doctor 2	4	1	1	4	7	7	7	7

Table 5.2: Preferences of the doctors.

5.3 Secure Equitable Cake-Cutting

This section presents the secure protocol for an equitable cake-cutting between two parties and gives implementation results. As described in Section 2.5, we build our protocol on top of the arithmetic black-box functionality $\mathcal{F}_{ABB}$ of Damgård and Nielsen [51] extended by Toft [52]. Let q be a prime and let $l \in \mathbb{Z}$ be a complexity parameter with $l := \lceil \log_2 q \rceil$. The notation $[x]_q$ represents a share of x over $\mathbb{Z}_q$ and the notation $[u_m]_q$ stands for $[u_1]_q, \ldots, [u_N]_q$.

Our protocol requires a third partly-trusted player. He is not a mediator in the strict sense because he does not know the sensible data: the utilities u_m and v_m. He gets only shares of these values. So, the two players need to rely less extensively on his integrity. The third partly-trusted player has to follow the protocol like any other player.

The two building blocks of the cake-cutting protocol are the bit decomposition and the approximate inversion protocols. The bit decomposition protocol is the key tool to securely compare two shared secrets [57]. It computes the bit-decomposition $a_0, \ldots, a_{l-1} \in \{0,1\}$ of $a = \sum_{i=0}^{l-1} a_i 2^i$ with $a \in \mathbb{F}_q$. The total complexity is 114 rounds and $110l \log_2 l + 118l$ invocations of the multiplication protocol.

The approximate inversion protocol distributively computes a floating point approximation of $1/p$ using the Newton iteration [86]. Inputs are polynomial shares of p and outputs are polynomial shares of an integer $\tilde{p}$ so that $\tilde{p}/2^{t+l} = 1/p + \epsilon$, where $0 < \tilde{p} < 2^{t+2}$ and $|\epsilon| < (k+1)2^{-l-t+4}$. To have the r most significant bits of $1/p$ and $\tilde{p}/2^{t+l}$ equal, the parameter t must be chosen bigger than $r + 5 + \log_2(k+1)$. The total complexity is $\mathcal{O}(\log_2 t)$ rounds and $\mathcal{O}(\log_2 t)$ invocations of the multiplication protocol.

5.3.1 Protocol for 2 Players

We present a secure protocol to compute an equitable division for two players that relies on the procedure described in Section 5.2.1. The equitable cutting point is computed in 2 steps. First, Protocol 5 determines the interval in which the cutting point is included. Then, Protocol 7 computes the exact value of the cutting point. We detail the protocols hereunder.

Protocol 6 securely computes the coefficients $a_i = u_i + v_i$ and $b_i = -\sum_{m=1}^{i} u_m + \sum_{m=i+1}^{N} v_m + i\,(u_i + v_i)$ for the interval $]i-1, i]$. It does not require any communication between the players since u_m and v_m are polynomial shared for all m. Multiplication mod q of a shared element and a known element of $\mathbb{Z}_q$ (`locmul`) is achieved by having all parties locally multiply (mod q) the known element by their shares.

Protocol 5 determines the interval in which the cutting point is included. It is recursively called up to $\log_2(N)$ times. To check if the cutting point is in the interval $]i-1, i]$, we test whether $b_i \in](i-1)a_i, ia_i]$. It is equivalent to test whether $b_i/a_i \in]i-1, i]$ and does not require a secure division. Lines 3 to 9 perform a secure comparison of the shared coefficient b_i with the bound of the interval multiplied by a_i. Protocols from Damgård *et al.* enable unconditionally secure constant round multi-party computation for comparison [57]. The comparison function is $<^?\colon \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{Z}_q$, where $(x <^? y) \in \{0,1\}$ and $(x <^? y) = 1$ iff $x < y$.

The complexities of the protocols `bit-lt` and `bits` are respectively 19 rounds and $22l$ invocations of the multiplication protocol and 114 rounds and $110l \log_2 l +$

$118l$ invocations of the multiplication protocol [57]. Protocol 5 uses 3 invocations of `bits` and 2 invocations of `bit-lt` giving a total complexity of 133 rounds and $330l \log_2 l + 398l$ invocations of `mul`.

Let us consider the case of 2 players and a third partly-trusted player (k=3) with a utility function of N=10 steps of length $\ell_x = 4$, where we only want to find the correct interval, that is, we do not use the `cutting-point` protocol. In the worst case, the protocol `interval` is called $\lceil log_2 N \rceil = 4$ times giving a complexity of 532 rounds and 16928 invocations of the multiplication protocol.

Protocol 5: `interval`

Input: The list of secret utilities of both players $[u_m]_q, [v_m]_q$ and the public left/right bound of the "cake" interval c/d.

Output: The approximate cutting point $\tilde{\alpha}$.

$i = \left\lceil \frac{c+d}{2} \right\rceil$;
$([a]_q, [b]_q) \leftarrow$ `coefficients`$([u_m]_q, [v_m]_q, i)$;
$[g]_q \leftarrow$ `locmul`$(i-1, [a]_q)$;
$[h]_q \leftarrow$ `locmul`$(i, [a]_q)$;
$[g]_B \leftarrow$ `bits`$([g]_q)$;
$[h]_B \leftarrow$ `bits`$([h]_q)$;
$[b]_B \leftarrow$ `bits`$([b]_q)$;
$[x]_q \leftarrow$ `bit-lt`$([b]_B, [h]_B)$;
$[y]_q \leftarrow$ `bit-lt`$([g]_B, [b]_B)$;
$x \leftarrow$ `reveal`$([x]_q)$;
$y \leftarrow$ `reveal`$([y]_q)$;
if $x + y = 2$ **then**
  $\tilde{\alpha} \leftarrow$ `cutting-point`$([a]_q, [b]_q)$;
  return $\tilde{\alpha}$;
end
if $x = 1$ **then**
  `interval`$([u_m]_q, [v_m]_q, c, i-1)$;
end
if $y = 1$ **then**
  `interval`$([u_m]_q, [v_m]_q, i, d)$;
end

Protocol 7 computes the cutting point $\alpha_i = F_i(u_m, v_m) = b_i \; a_i^{-1}$. Protocols from Algesheimer *et al.* enable to compute the exact value of the cutting point [86]. The protocol `appinv`$([a]_q)$ inverts the polynomial shared coefficient a. It is iterated $\mathcal{O}(\log_2 t)$ times and one iteration requires 12 rounds and 2 invocations of `mul`. Finally, `appmul`$([\tilde{a}]_q, [b]_q)$ computes shares of an approximation to α_i. It requires 6 rounds and 1 invocation of `mul`. The protocol `cutting-point` only deals with integers. The "tilde" variables approximate non integers values (see

Protocol 6: `coefficients`

Input: The list of secret utilities of both players $[u_m]_q, [v_m]_q$ and the interval i.
Output: The coefficients $[a]_q$ and $[b]_q$.

$[a]_q \leftarrow \texttt{add}([u_i]_q, [v_i]_q)$;
$[p]_q \leftarrow \texttt{locmul}(i, [a]_q)$;
for $m \leftarrow 1$ **to** i **do**
  $[-u_m]_q \leftarrow \texttt{locmul}(-1, [u_m]_q)$;
end
$[b]_q \leftarrow \texttt{add}([-u_1]_q, ..., [-u_i]_q, [v_{i+1}]_q, ..., [v_N]_q, [p]_q)$;
return $([a]_q, [b]_q)$;

Protocol 7: `cutting-point`

Input: The coefficients $[a]_q$ and $[b]_q$.
Output: The approximate cutting-point $\tilde{\alpha}$.

$[\tilde{a}]_q \leftarrow \texttt{appinv}([a]_q)$;
$[\tilde{\alpha}]_q \leftarrow \texttt{appmul}([\tilde{a}]_q, [b]_q)$;
$\tilde{\alpha} \leftarrow \texttt{reveal}([\tilde{\alpha}]_q)$;
return $\tilde{\alpha}$;

Table 5.3).

Variable	Approximated value
$\tilde{a}$	$2^{t+l}/a$
$\tilde{b}$	$b/2^r$
$\tilde{\alpha}_i = \tilde{a} \cdot \tilde{b}$	$\alpha_i \cdot 2^{t+l-r}$

Table 5.3: Variables and corresponding approximated values used in Protocol 7.

Let us go back to the example of 2 players and a 10-step utility function. We would like to compute the complexity of the `cutting-point` protocol. To have the $r = 4$ most significant bits of $1/a$ and $\tilde{a}/2^{t+l}$ equal, the parameter t must be chosen bigger than $r + 5 + \log_2(k + 1)$. The protocol `appinv` is thus iterated at least $\lceil log_2 7 \rceil = 3$ times and its total complexity is 37 rounds and 6 invocations of `mul`. The total complexity of the `cutting-point` protocol is 43 rounds and 7 invocations of `mul`. The total complexity of the protocol `interval` (with the computation of the cutting point) is approximately 700 rounds and 17000 invocations of the multiplication protocol.

5.3.2 Protocol for k Players

While there was a unique cutting point with two players, there are $k!$ distinct vectors of cutting points leading to a different equitable division with k players. Indeed, the $k!$ possible orderings of the players lead each to a different system of equations that has a unique solution since the matrix A is always invertible (Proof in Section 5.2.2).

The protocol to compute an equitable division for a chosen players ordering is quite similar to the protocol for two players. Nevertheless, an equitable division achieved through an arbitrary ordering is not necessary proportional. In other words, the k players have no guarantee to receive at least $1/k$ of the cake according to their valuation. Securely computing the most efficient division among all equitable divisions for k players would then require to compare the outcome of the $k!$ divisions resulting from the players' permutations.

5.3.3 Implementation Results

The protocol for two players is implemented thanks to the Virtual Ideal Functionality Framework (VIFF). While VIFF can be used for applications in the passive as well as in the active model, we only implemented our protocol with security against a passive adversary.

Let us go back to the application of two doctors on duty who wish to determine their respective shift (Section 5.2.3). A straightforward implementation runs in about 20 seconds (with 7 seconds dedicated to the division needed for computing the final cutting point). The result is $\alpha = 6.684$, which means that the two shifts are respectively from 6 am to 12:40 am and from 12:40 am to 9 pm. The first doctor has a utility of 39,3 for the first shift and a utility of 60,7 for the second one and vice-versa for the other doctor, so the first doctor will take the morning shift while the second one will take the afternoon shift.

We tested the efficiency of our protocol for different numbers of intervals. The following table shows the time taken according to the number of intervals.

Number of intervals	5	10	15	20
Execution times (sec)	13	16	19	22

Table 5.4: Execution times in function of the number of intervals.

5.4 Conclusion

This chapter presents an equitable cake allocation procedure suitable for the existing MPC techniques. This procedure is translated into a secure protocol that relies on existing MPC techniques.

The cake-cutting problem is modelled to allow the use of MPC protocols. We model the players' preferences by step functions. This class of functions is quite appropriate to MPC. Furthermore, a lot of functions can be approximated by step functions. Adapting the size of the cake to the number of steps of the function makes it possible to deal with integer values as long as possible.

The cake allocation procedure gives an equitable division with uncut pieces for two and k players. Step functions allow finding an equitable division by only solving linear systems. For two players, there is only one envy-free and equitable division. It is found by solving $\log_2 N$ linear equations. This complexity is obtained through a binary search algorithm. The secure MPC protocol consists of two different stages for two players. First, a solution is found in a given interval. Then, this solution is compared to the bounds of the interval to determine if it solves the cake-cutting problem. There are $k!$ possible equitable allocations for a division between k players. Each equitable division is obtained by solving Γ_N^{k-1} linear systems.

The equitability criterion implies dealing with non integer cutting points in the secure protocol. This is not convenient because it is not easy to solve a system with non integer solutions thanks to MPC. An almost equitable division in which only integer cutting points are allowed seems to be a good alternative. It could be all the more interesting with large step functions.

It is interesting to notice that, in this case, the best way to achieve a secure protocol is to design a completely new procedure. This highlights a completely different behaviour compared to the sorting network approach, for example. If we compare our oblivious protocol with its hypothetical non-oblivious counterpart, there is no overhead in term of asymptotic complexity. However, oblivious operations like comparisons are still more costly than non-oblivious one.

The cutting points of an equitable division reveal a precise information about the players' preferences. Equitability requires that all the players receive exactly the same proportion of the cake according to their own valuation. An envy-free division does not reveal such an exact information. Envy-freeness means that any player prefers his own share to all the other ones. It could be interesting to explore this criterion, which has not been developed here. An interesting perspective would be to use Su's algorithm which relies on a combinatorial

result known as Sperner's lemma [96]. This infinite algorithm constructs an envy-free division by successive approximations [97].

There are two directions to bridge cryptography and game theory. We developed only one direction: designing a new cake allocation procedure to solve the game-theoretic problem of fair division under the cryptographic settings. The other direction is to apply game theory to cryptography. In the game-theoretic settings, all players are supposed to be rational, that is, acting in their own and best interest. In this case, Shamir scheme is no longer relevant. An interesting perspective would be to use the concept of rational secret sharing [98], [99] and rational protocol design [100], [101].

Chapter 6

Securely Solving Simple Combinatorial Graph Problems

Graphs are one of the first objects of study in discrete mathematics. A graph consists of *vertices* (also called *nodes*) and of *edges*. The edges are the connections between the vertices. An edge may be oriented and may also have a cost. Modelling road maps is easy thanks to graphs. A city becomes a vertex and a road an edge. The length of the road is the cost associated to an edge. Besides road maps, graphs may be used to represent things as various as networks of communication, computational devices, social networks, molecules or migration paths.

Well-known graph algorithms exist to compute various problems. For example, the minimum spanning tree (the cheapest sub-graph where each pair of vertices is connected by exactly one simple path), the point-to-point shortest path (the shortest path between two vertices) or the maximum flow between two vertices.

The goal of this chapter is to present a way to solve simple combinatorial graph problems in a secure multi-party setting. Our protocols can be used as such or as building blocks for more complex secure applications. The protocols rely on previous work on MPC and on traditional graph algorithms.

Section 6.1 describes our contributions, related works as well as our modelling and implementation choices: the graph representations and the minimal bounds (on the size of $\mathbb{Z}_m$ for example). Section 6.2 describes our approach to the

classical single-source shortest path problem. Section 6.3 describes our approach to the maximum flow problem. Finally, Section 6.4 raises some open problems.

6.1 Preliminaries

One common point of the applications developed in Section 2.4.1 is that the function evaluation process is naturally oblivious of the inputs on which the function has to be evaluated. Computing the highest of n bids or summing n votes is carried out by performing n comparisons or sums independently of the values that are considered. There are large classes of problems however for which the natural evaluation process depends on the input data. In that case, even if all the manipulated data are appropriately shared or encrypted, the execution flow might just be sufficient to leak undesirable information.

This is typically the case in combinatorial problems, of which graph problems are one of the most common examples. Consider, for instance, a consortium of delivery companies covering different territories through regular distribution circuits. These companies might be interested in computing the fastest way to bring a package from one place to another, but be reluctant to share with each other the precise connections they use and the performance of their trucks. Their problem could be solved by securely evaluating traditional shortest path algorithms such as those of Bellman-Ford or Dijkstra.

The immediate way of securely computing the shortest path would be to blind (encrypt or share) the weight of all the edges of the corresponding graph. However, this approach could completely miss its purpose depending on the graph encoding and shortest path algorithm that are used: if the algorithm conditionally visits the graph by branching as a function of the secret weights, then the branching patterns could leak a substantial amount of secret information. In a similar way, the resolution of combinatorial problems, even on obfuscated inputs, can leak substantial information through the structure of the combinatorial object that is manipulated, as well as through its running time. We stress that this is not just a theoretical concern: numerous techniques have been developed, notably in the line of work on side-channel attacks [102], that can successfully exploit branching patterns and running times in order to recover the secrets on which computation is performed.

6.1.1 Our Contributions

This chapter investigates the problem of securely solving combinatorial problems in a multi-party setting through series of examples taken from graph theory. To the best of our knowledge, it was the first time that these most classical algorithmic problems had been addressed in a general secure multi-party setting. Our solutions have applications in the numerous contexts where a graph is shared between competing entities. Natural examples include:

- Secure GPS guidance in which one party knows the map while the other knows his origin and destination.
- Secure determination of topological features in social network (the number of different ways to connect two people can be seen as a special case of the maximum flow problem, for instance, in which case each party would know his own friends but no more).
- Secure determination of the performance of the cooperation between competing network operators (gas, electricity, logistics, ...), in which each party would know the capacity of his own infrastructure but no more.

Furthermore, our study raises several intriguing complexity gaps and suggests the exploration of various trade-offs. Parts of this work were written in collaboration with A. Aly, E. Cuvelier, O. Pereira and M. Van Vyve and presented at Financial Cryptography 2013 [2].

Algorithm Design. We focus our research on computing the shortest path and the maximum flow based on the secure arithmetic black-box functionality of Damgård and Nielsen [51] augmented with comparison [52]. That is, our protocols assume access to a functionality that offers secure addition, multiplication and comparison. This allows us to abstract from the specific security model in which we want our protocol to be secure: depending on the implementation of the secure arithmetic black-box that is used, our protocols will be secure only in the presence of an honest majority or with up to all but one corrupted player, in the information theoretic or computational model, in front of passive or active adversaries, ... As described in Section 2.4.2, various such implementations, in various models, are available in tools designed for multi-party computation such as FairplayMP [42], Sharemind [43], Sepia [45] or VIFF [48]. The implementation of our prototypes is realized thanks to this last framework.

We focus on two of the most standard graph problems, chosen for their wide diversity of applications: computing shortest paths and maximum flows. For each of these problems, we discuss secure evaluation techniques inspired from classical algorithms of various complexities: Bellman-Ford and Dijkstra for the

shortest path, and Edmonds-Karp and Push-Relabel for the maximum flow.

Our resulting algorithms offer quite different overheads, depending on the algorithm and the graph structure, as illustrated in Table 6.1. For those algorithms, the table shows first the traditional (non secure) complexity, then the complexity of our secure versions expressed in number of calls to the arithmetic functionality. There, we consider the case of a graph with public structure and then with private structure, meaning that not only the weight of each edge is kept secret, but that the adjacency relation between vertices is kept private as well.

Several observations can already be made.

- The best implementations, using advanced data structures as dynamic trees [103] or Fibonacci heaps [104], are definitely non-trivial to replicate in the secure setting (see also discussion in Section 6.1.2 below). Their relevance is also unclear for the relatively small size of the problems that we are addressing, as they usually come with large constants.
- The overheads resulting from moving from the original algorithms to their secure versions largely differ between algorithms: in the case of a public structure for instance, we see either no difference, or an $|E|$ factor or a $|V|$ factor depending on the algorithm.
- The overhead resulting from hiding the graph structure largely differs depending on the algorithm and type of graph. For Bellman-Ford and Push-Relabel, the difference essentially corresponds to always handling a complete graph when the structure needs to be hidden. For Dijkstra however, the secrecy of the graph structure has no impact.
- While Bellman-Ford is traditionally less efficient than Dijkstra, this is no longer true (asymptotically at least) for our secure variants: Bellman-Ford becomes substantially more efficient for sparse graphs (e.g., if $|E| = \mathcal{O}(|V|)$) and the asymptotic complexities are similar for dense graphs.

The overheads in terms of number of protocol participants, round complexity, ... largely depend on the implementation of the secure arithmetic functionality, and are in line with traditional works.

Complexity: The Constants Matter. In order to challenge our algorithms in practice, we implemented them all using the Virtual Ideal Functionality Framework (VIFF) of Geisler *et al.* [48], in the honest-but-curious model.

This allowed us to further investigate the constants hidden by the asymptotic notations discussed above. This made particularly visible the difference of cost

	Optimized	Original	Public Structure	Secret Structure
Bellman-Ford	$\|V\|\|E\|$	$\|V\|\|E\|$	$\|V\|\|E\|$	$\|V\|^3$
Dijkstra	$\|E\| + \|V\| \log \|V\|$	$\|V\|^2$	$\|V\|^3$	$\|V\|^3$
Edmonds-Karp	$\|V\|^2\|E\|$	$\|V\|\|E\|^2$	$\|V\|\|E\|^2$	$\|V\|^5$
Push-Relabel	$\|V\|\|E\| \log(\frac{\|V\|^2}{\|E\|})$	$\|V\|^3$	$\|V\|^2\|E\|$	$\|V\|^4$

Table 6.1: Asymptotic complexities: original algorithms and secure versions with public and private graph adjacency matrix.

between the different black-box primitives that we used: addition based on linear secret sharing [14] comes for free (no communication involved), multiplication is noticeable (it involves one secret sharing), and comparison (based on Toft's protocol [105],[81]) is ≈ 165 times more expensive than a multiplication, something that strongly contrasts with the execution time of traditional algorithms.

These differences have strong practical impact and motivated some trade-offs as well.

- Our version of Dijkstra's algorithm only involves $|V|^2$ comparisons compared to $|V|^3$ (or $|V||E|$) in Bellman-Ford. As a result of this, for dense graphs or when the graph structure is secret, Dijkstra's algorithm remains considerably more efficient than Bellman-Ford's, even when the structure of the graph is public, provided that the graphs have a reasonably small size (a hundred vertices).
- For sparse public graphs that contain a small number of paths from the source to the sink, a variant of the Edmonds-Karp's algorithm that relies on an exhaustive public enumeration of the source to sink paths can be considerably simpler and more efficient than a secure version of the breadth-first search for augmenting paths that is performed in the original algorithm: this allows trading expensive book-keeping and addressing operations for more but much simpler rounds.

So, besides the fact that our work offers the first solutions for the secure evaluation of various graph properties, we think that it raises several intriguing complexity issues. Notably, we wonder whether the complexity gaps that we have are inherent to the added security or if they can be improved.

6.1.2 Related Works

Graph Theory. An important literature on finding the shortest path in a graph exists. Dijkstra's algorithm computes the shortest path tree in a graph with positive weights [106]. Sedgewick and Vitter present an algorithm to compute the shortest path in a euclidean graph [107] while Fredman and Tarjan propose an improvement based on Fibonacci heaps [104]. To the best of our knowledge, there was no known result about MPC applied to graph theory in 2013. Nevertheless, Attalah and Du mention the shortest path problem as an open problem for secure multi-party computation [108].

As mentioned above, the large majority of works on secure multi-party computation focused on functions whose evaluation execution flow is independent of the secret inputs. There are important exceptions to this, however.

Branching Programs. Branching programs are decision procedures that, based on some inputs and decision parameters, such as thresholds, perform a specific classification of the input. Secure versions of these programs, where a user does not learn the branching program of the server while the server does not learn the user's inputs, have been considered in various works [109], [110], [111], [28], [112]. While these works share our goals of hiding the data path through which the program is going, they do not aim at hiding the length of that path which, in our case at least, could leak a substantial amount of information.

Shortest Path In The Two-Party Setting. Brickell and Shmatikov addressed the problem of securely solving some graph problems and their work is, as such, the closest to ours [113]. Substantial differences appear, though.

First, their security model is quite different from ours. Their protocols, which are based on a secure set union protocol, proceed by progressively making their outputs known to the participants as part of the execution (e.g., edge by edge as the protocol runs). Even though this is not revealing more than the eventual outcome, this makes their protocols unusable as sub-components of other higher-level protocols that would rely on using these outputs as part of their secret state. Revealing outputs part-by-part as the protocol runs might also be problematic in applications in which some participants could abort the protocol in the middle of its execution, based on what they have already learned. Our protocols, on the other hand, can be freely used as subroutines, and one of our secure max-flow algorithms will make use of a secure shortest-path algorithm.

Second, the graph problems they consider are different from ours as well. They do not consider the maximum flow problem at all: their work focuses on

computing the shortest distances, from a known source to all the vertices or for all the vertex pairs, in a setting where all the participants assign a weight to all the edges. We further investigate the problem of computing the shortest path from a single source to a single destination, which cannot be done using their set union technique as it would reveal much more information than the specific distance we are interested in.

Eventually, their protocols are not based on generic building-blocks, like the arithmetic black-box functionality on which we rely. Specifically, their protocols are designed for the two-party computation setting in the honest-but-curious model. While these specifics allow them to develop techniques that are quite efficient in this two-party setting, it is unclear how efficient a transposition of their approach to the multi-party setting would be.

Efficient Secure Data-structures. The problem of securely computing on data-structures has recently been investigated by Toft [52], in the case of a secure priority queue, which he implements using a variation of bucket heap. The problem studied there shares similar flavours with those we address here: to securely compute on structured data by keeping the actions independent of the inputs. The computational overhead compared to the efficiency of the original bucket heap is logarithmic, making it occupy an interestingly different spot in the list of overhead examples discussed above. Detailed background on data structures can be found in standard references including Knuth [114].

Oblivious memories. In their recent paper, Efficient, Oblivious Data Structures for MPC [115], Keller and Scholl present, among others, an oblivious implementation of a priority queue with only poly-logarithmic overhead compared with the classical counterpart. The oblivious priority queue is then used for an implementation of Dijkstra's shortest path algorithm on general graphs with secret structure. The complexity is $\mathcal{O}(|E| \log^5 |E| + |V| \log^4 |V|)$, with $|V|$ and $|E|$ public.

Keller and Scholl benchmarked their protocol on cycle graphs, that are graphs where the edges form a cycle passing every vertex exactly once. Their solution outperforms ours for these low-degree graphs, especially with graphs counting more than 128 vertices. However, our solution is always more efficient with complete graphs. For a 128-vertex complete graph, their proposed solution takes about 10^5 seconds while their implementation of our solution only takes about 500 seconds.

6.1.3 Modelling and Implementation Choices

Graph Representation. Depending on the algorithm we are trying to compute and on the part of the graph description that is part of the secret input, different graph representation approaches will be useful.

In all cases, we will assume that the number of vertices in the graph is public (or at least an upper bound on it). Depending on the setting, the adjacency relationship between the vertices might be public or not. For instance, it is natural to have it public if the graph represents the connections between places on a map, but it might be desirable to keep it secret if the presence of edges reveals the existence of transactions between competing companies.

A traditional structure for storing a graph consists in storing, with every vertex, a list of its neighbours (and the weight of the corresponding edges). This structure is quite efficient in terms of memory. However, it might be quite problematic from a security point of view, as it discloses the degree of each vertex. A solution would be to tolerate the leakage of an upper bound on these degrees, but that upper bound would be close to imply the storage of a complete graph as soon as one single vertex is of high degree. Furthermore, even if the leakage of the degree of the vertices is tolerated, algorithms that perform breadth-first search on vertices and branch depending on the weight of edges could reveal a lot of information. As a result, this graph representation can be very effective in some cases, but completely inappropriate in others, even when the graph structure is public.

A second traditional way of representing graphs is to store their adjacency matrix, the elements of the matrix representing the weight of the edges between vertices. This approach has the benefit of offering a storage that is independent of the graph structure. While running our algorithms, we will often need to perform some operations on a specific vertex designated by a secret index. This will typically be performed by running that operation on all vertices, including a cancelling factor everywhere but on the vertex that needs to be treated. An obvious way of testing whether we are working on the right vertex would be to perform a test at each step. We actually use a more effective approach by representing the index of vertex i by a vector $[\mathbf{i}] \in \{[0],[1]\}^{1\times n}$ where each entry is [0] except for the i'th which is [1]. We can then access the weight of the edge from vertex i to vertex j by computing the matrix product $[\mathbf{i}].[\mathbf{W}].[\mathbf{j}]^t$. Chapter 3 explains this unary approach in details.

For a graph with n vertices, Protocol 1 (extended for matrices) allows retrieving a secret position in the adjacency matrix in $\mathcal{O}(n^2)$ multiplications instead of $\mathcal{O}(n^2)$ comparisons, which is considerably more efficient, even if it implies a

Protocol	$\top >$	$m >$
SSSP1	$\lvert V\rvert \cdot \mathbf{w}$	$f(\top)$
SSSP2	$\lvert V\rvert \cdot \mathbf{w}$	$\lvert V\rvert \cdot f(\top)$
SSMF1	-	$\lvert V\rvert \cdot f(\mathbf{c})$
SSMF2	-	$\texttt{max}(2\lvert V\rvert, \lvert V\rvert \cdot f(\mathbf{c}))$
SSMF3	$\mathbf{c}$	$\texttt{max}(f(\top + \mathbf{c}), \lvert V\rvert \cdot f(\mathbf{c}))$

Table 6.2: Minimal bounds on $\top$ and m to avoid overflows.

considerable overhead in storage (moving from 1 secret index to n secret bits). We note that, in all cases, this approach implies treating the graph as if it were complete, which can be a considerable waste of resources if the graph is actually sparse.

Bounds. The size of the ring $\mathbb{Z}_m$ has to be chosen carefully to prevent overflows. For each protocol presented in this chapter, we provide the bounds of m and the value of $\top$ in Table 6.2. These bounds depend on numbers such as the maximum weight $\mathbf{w}$ or the maximum capacity $\mathbf{c}$ allowed for the edges. These maxima are agreed in advance by the players. Remark that $\top$ is smaller than m. Most comparison protocols require a much larger m than the values to compare. This dependence is taken into account via a function f.

6.2 Secure Shortest Path Problem

The single-source shortest path problem is a major problem in graph theory. It has several immediate applications. The typical one is finding the shortest way to connect two cities on a road map where each city is represented by a vertex and each road between two cities by an edge. The edge weights are the road distances between cities. In this context, a user may then want to obtain driving directions without revealing neither his starting point nor his destination. If multiple entities maintaining each different roads in a network want to compute the cheapest possible path between two locations, they need to use a secure computation of the shortest path too.

Another application is the one of two entities owning each a secret location in a shared network and willing to compute the distance between them without disclosing their location. We note that such a problem is worth solving even for relatively small graphs. Consider for instance a routing network with a dozen

hubs in different European countries and three competing logistic companies having each their own transportation costs for a defined set of roads. As costs typically represent sensitive information that should not be disclosed to competitors, being able to securely solve the shortest path problem for 3 parties and a graph with a dozen of vertices is quite helpful. Similar problems happen for network traffic on routers where a small number of big hubs is involved. Competing companies have to solve the shortest path to define routing schemes without revealing sensitive information about internal network configuration.

Shortest path algorithms are also used as sub-algorithms for more advanced problems like the maximum flow problem that we address in Section 6.3 or the Chinese postman problem [116] [117]. This last problem consists in finding the shortest cycle going through every edge at least once. Finding an optimal solution is NP-hard. A traditional solution to the Chinese postman problem makes use of a shortest path algorithm to determine which edges have to be visited twice: it computes all the shortest paths between the odd degree vertices [118]. This highlights again the importance of keeping our protocols composable.

We investigate two standard algorithms for finding the single-source shortest path in a graph with weighted edges: Dijkstra's algorithm and Bellman-Ford's algorithm. The first one requires all edge weights to be positive, while the second one only assumes there is no negative-weight cycle in the input graph. As the non-secure version of all the algorithms that we treat is widely available [119], [120],[62], we will only briefly outline them.

All our protocols assume that inputs are already stored in the $\mathcal{F}_{ABB}$ functionality and give access to the stored outputs (that can be opened through opening requests to $\mathcal{F}_{ABB}$). This feature guarantees the composability of the protocols. The way inputs and outputs are shared depends on the application: they might come from a specific problem, or from the needs of a higher-level protocol using this protocol as a sub-routine, for instance.

6.2.1 Bellman-Ford's Algorithm

The algorithm of Bellman-Ford is particularly simple, making it a natural target for building a secure version [121]. This algorithm proceeds by repeatedly scanning all edges, in search of adding edges that decrease the ongoing distance from the source to the various vertices. If a pass over the edges did not improve the current solution, or if the edges were scanned $|V|$ times, the algorithm halts. An interesting feature of this algorithm is that its flow of operations only depends on the structure of the graph but not on the weight of the edges. Its

drawback is its time-complexity: its classical implementation runs in $\mathcal{O}(|V||E|)$ time.

Protocol 8 (the SSSP1 protocol) presents our secure shortest path protocol based on Bellman-Ford. Note that $h(e)$ and $t(e)$ represent the head and tail vertex of an edge e respectively. As discussed in Section 6.1.3, note that $\top$ is a number agreed in advance by the players as a higher bound for some calculations of the protocol. Finally, note that `updatevector` refers to Protocol 1. The SSSP1 protocol differs from the original algorithm only in a limited number of aspects:

- The branching corresponding to the discovery of a shorter path is handled on Lines 8–10 through arithmetic operations as in Protocol 1.
- The early termination condition of the Bellman-Ford algorithm, which is triggered if the inner loop happens to have no effect during one pass, is removed as it could leak information. This does not invalidate the correctness of the algorithm but only increases the running time.

Protocol 8: SSSP1 protocol based on Bellman-Ford's algorithm

Input: A graph $G = (V, E)$ where V is the set of vertices and E the set of edges, a vector of the shared weights $[\mathbf{W}]_e$ for each $e \in E$, and a share of the source vertex $[s] \in V$.

Output: The list of immediate predecessors $[\mathbf{P}]$ and/or the list of total distances $[\mathbf{D}]$.

1 **for** $i \leftarrow 1$ **to** $|V|$ **do**
2 $\quad [\mathbf{P}](i) \leftarrow [0]$;
3 $\quad [\mathbf{D}](i) \leftarrow [\top]$;
4 **end**
5 `updatevector`$([\mathbf{D}], [s], [0])$;
6 **for** $i \leftarrow 1$ **to** $|V|$ **do**
7 $\quad$ **for** $e \leftarrow 1$ **to** $|E|$ **do**
8 $\quad\quad [y] \leftarrow [\mathbf{D}](t(e)) - [\mathbf{D}](h(e)) + [\mathbf{W}](e)$;
9 $\quad\quad [x] \leftarrow [y] < 0$;
10 $\quad\quad [\mathbf{D}](h(e)) \leftarrow [\mathbf{D}](h(e)) + [x] \cdot [y]$;
11 $\quad\quad [\mathbf{P}](h(e)) \leftarrow ([1] - [x]) \cdot [\mathbf{P}](h(e)) + [x] \cdot t(e)$;
12 $\quad$ **end**
13 **end**
14 If there was an update during the very last pass, the solution is unbounded (there is a negative cycle).

The structure of this algorithm makes it easy to implement with either of the two graph representations discussed above (list or matrix). It is thus possible

to fully exploit the sparsity of the graph when the structure is public (we use the matrix representation if it has to be kept secret).

It can be seen that our implementation requires $|V||E|$ secure comparisons, dominating the time required to perform $2|V||E|$ secure multiplications and $5|V||E|$ additions. These complexities grow to $\mathcal{O}(|V|^3)$ when the graph structure is secret, as the graph is then treated as complete (i.e., augmented with edges of infinite weight). Very interestingly, this algorithm is the only one among those we treated in which our solution does not raise any asymptotic overhead (when the structure is public).

Security. The simulation of an execution of this protocol is immediate from the simulators available for the different calls that can be made by the $\mathcal{F}_{ABB}$ functionality: the simulators corresponding to each of the '$+$', '$\cdot$' and '$<$' operations can be invoked in turn, in an order defined by the protocol execution, and a number of times that only depends on public values ($|V|$ and $|E|$).

Proposition 3. *The protocol is secure if a player is not able to distinguish an execution of the protocol in the real world (where he is exchanging information with the other players) from an execution in the ideal world (where he is exchanging information with a trusted party via a simulator).*

Proof. The simulators of the basic operations are invoked in a predefined order that does not change from one execution to the other. For example, in Protocol 8, there are $|V| \cdot |E|$ calls to the sequence: 2 '$+$', 1 '$<$', 1 '$\cdot$', 2 '$+$', 2 '$\cdot$' and 1 '$+$'. This sequence is constant and is always applied on the same shares. The number of calls to the sequence only depends on the number of edges and vertices in the graph. □

The same argument will apply to the other protocols we present in this chapter, and we will therefore not come back to it.

6.2.2 Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest path from the source to all vertices in the graph, that is, the shortest path tree rooted at the source. The algorithm is greedy. At each iteration one vertex (the one with the smallest distance label) is permanently updated to the status *scanned*. This feature may be exploited to reduce the number of iterations if we are only interested in the shortest path from the source to a given target and not in a complete shortest path tree. The

computations can be stopped once the target vertex has been scanned, even though this may leak information about the weights.

Adapting Dijkstra. The fact that Dijkstra's algorithm goes through the graph in an order that depends on the weight of the edges makes it very difficult to efficiently exploit the sparsity of a graph: our best solutions have all a complexity that amounts to the one of a complete graph, and we therefore use the matrix representation in all cases for our protocol.

Protocol 9: SSSP2 protocol based on Dijkstra's algorithm.

Input: A graph $G = (V, E)$ where V is the set of vertices and E the set of edges, a matrix of shared weights $[\mathbf{W}]_{i,j}$ for $i, j \in \{1, ..., |V|\}$ and a source vertex $[\mathbf{s}] \in V$ (given in unary representation).

Output: The vector of distances $[\mathbf{D}]_i$ for $i \in \{1, ..., |V|\}$ and the matrix of predecessor $[\mathbf{P}]_{i,j}$ for $i, j \in \{1, ..., |V|\}$.

for $i \leftarrow 1$ **to** $|V|$ **do**
  $[\mathbf{D}](i) \leftarrow [\top]$;
  $[\mathbf{Q}](i) \leftarrow [0]$;
  for $j \leftarrow 1$ **to** $|V|$ **do**
    $[\mathbf{P}](i, j) \leftarrow [0]$;
  end
end
`updatevector`$([\mathbf{D}], [\mathbf{s}], [0])$;
for $i \leftarrow 1$ **to** $|V|$ **do**
  $[\mathbf{D'}] \leftarrow [\mathbf{D}] + [\mathbf{Q}]$;
  $[\min], [\mathbf{k}] \leftarrow$ `binarymin`$([\mathbf{D'}])$;
  `updatevector`$([\mathbf{Q}], [\mathbf{k}], [\top])$;
  for $j \leftarrow 1$ **to** $|V|$ **do**
    $[a] \leftarrow ([\mathbf{D}] + [\mathbf{W}](*, j)) \cdot [\mathbf{k}]$;
    $[c] \leftarrow [a] < [\mathbf{D}](j)$;
    $[\mathbf{P}] \leftarrow$ `updaterow`$([\mathbf{P}], j, [\mathbf{P}](j) + [c] \cdot ([\mathbf{k}] - [\mathbf{P}](j)))$;
    $[\mathbf{D}](j) \leftarrow [\mathbf{D}](j) + [c] \cdot ([a] - [\mathbf{D}](j))$;
  end
end
return $[\mathbf{D}], [\mathbf{P}]$;

Protocol 9 (the SSSP2 protocol) presents our secure shortest path protocol based on Dijkstra's algorithm. Note that `updatevector` refers to Protocol 1 and that `updaterow` is the natural extension of `updatevector` for replacing a complete row in a shared matrix. Protocol `binarymin` (Protocol 3) has been introduced by Toft to obtain the minimal value out of a vector of shared values. It securely computes a share of the minimal value, [min], along with a share of its index, $[\mathbf{k}]$. The protocol uses $\mathcal{O}(n)$ comparisons and multiplications. Its

overall round complexity is $\mathcal{O}(\log(n))$ rounds. Vector $\mathbf{Q}$ records the status of each vertex. An entry is equal to zero if the corresponding vertex has not been scanned yet. It is updated to $\top$ as soon as the vertex has been scanned.

The main differences between the traditional and our secure version of Dijkstra's algorithm happen in the inner loop:

- On Line 9, the loop goes through all vertices instead of only considering the neighbours of the current vertex. In particular, this includes an always transparent step where we consider the current vertex and gives a substantial overhead if a public sparse graph is considered.
- On Lines 4 – 8 – 12, we need to go through all elements of a row or a vector, even if we know that only one of them is going to be updated.

Those two modifications contribute to the same effect: they increase the original complexity of Dijkstra from $\mathcal{O}(|V|^2)$ to $\mathcal{O}(|V|^3)$. More precisely, the exact number of comparisons is $2|V|^2-3|V|+1$ and the exact number of dot products (used for the multiplication of vectors, costing $|V|$ multiplications) is $2|V|^2-|V|$ for $|V| \geq 4$.

As the comparison protocol we use requires 165 multiplications to compute a comparison, the number of multiplications to compute the shortest path in a complete tree is about $2|V|^3 + 329|V|^2 - 495|V| + 165$.

As illustrated in Figure 6.1, the switch from quadratic to cubic dominance is at about 165 vertices which is precisely the number of multiplications used by a single comparison.

Our secure version of Dijkstra comes with an overhead of a factor $|V|$ compared to the original one, even when the graph structure can be considered as public. We note that this was not the case in the work of Brickell [113] who considered running Dijkstra securely as well, but accepted to output the shortest paths step by step. Besides the limitation that this brings when the protocol has to be composed, we also observe that our algorithm can be used to solve problems that could not be solved by Brickell's approach, namely, computing the shortest path between two specific vertices without leaking any other information: their approach indeed leaks the shortest path to *all* vertices.

6.2.3 Implementation Prototypes

We implemented our protocols over the Virtual Ideal Functionality Framework to challenge their performance. We considered a 3-party execution in the

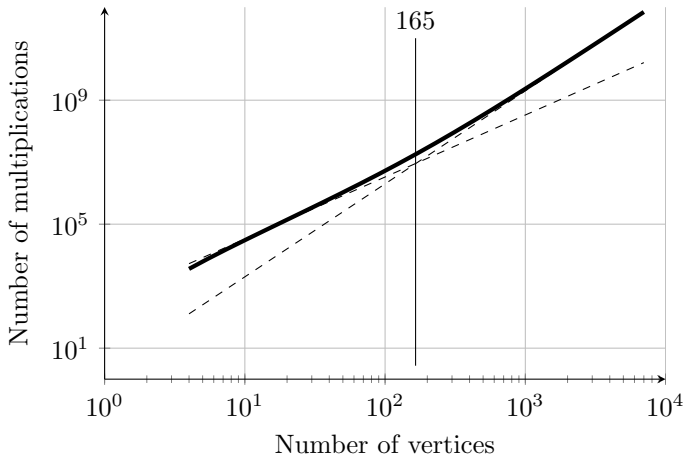


Figure 6.1: Number of multiplications when running Algorithm 9. The dashed lines highlight the quadratic then cubic growths.

information theoretic model with passive security: secret values are shared using Shamir's secret sharing, the BGW protocol is used for multiplication [5], and Toft's protocol is used for comparison [105]. These choices were made for simplicity and ease of prototyping, though much more efficient protocols exist and would have led to considerably shorter running times [53],[55]. The computation was performed on a single workstation equipped with an Intel Xeon CPUs X5550 (2.67GHz) and 24GB of memory, running a standard Debian Squeeze.

We ran the two shortest path protocols described above on complete graphs of various sizes. This first showed that Protocol 8 can only be conveniently used for graphs where $|V||E| \approx 10^3$ (a few minutes on a standard laptop): see Table 6.3.

Number of vertices	4	8	16	32	64	128
Execution times (sec) SSSP1	9	63	501	4003	31951	-
Execution times (sec) SSSP2	9	13	50	217	1018	5622

Table 6.3: Execution times of Protocols 8 and 9 for a complete shortest path tree.

Our secure versions of Bellman-Ford and Dijkstra have approximately the same

complexity for complete graphs. However the quadratic number of comparisons makes it possible to run our secure version of Dijkstra on a 64-vertex complete graph in roughly twice the time as taken by Bellman-Ford on a 16-vertex graph, and we have been able to run it up to a 128-vertex complete graph (i.e., counting 16256 directed edges) in a bit more than an hour.

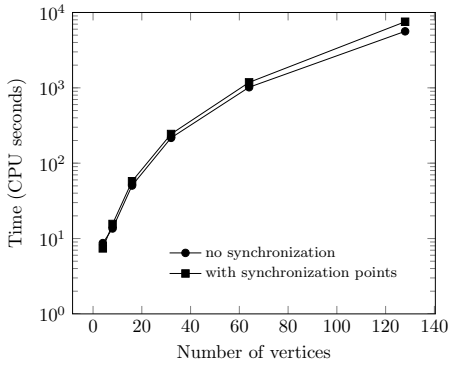
While these timings might look fairly high, they still make it possible to solve natural problems in a reasonable time. The 3-party, 12-vertex problem outlined above could be solved in about 30 seconds, for instance.

6.2.4 Comments on Memory Usage

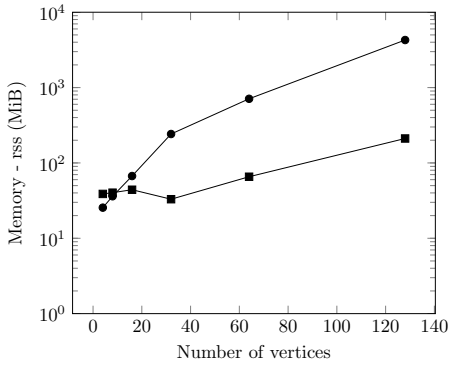
During our prototyping phase of Dijkstra's algorithm, we studied three different settings that differ with regards to the data considered as private. It results in a difference in terms of time and memory efficiency. All implementations use a matrix for the graph representation.

In the first implementation (the one described in Protocol 9), the graph and the source are shared. Dijkstra's algorithm is performed completely and outputs a vector with the shortest tree path from the source to all the other vertices. The execution time does not leak any information. The implementation does not use any synchronization point and is feasible up to 128 vertices.

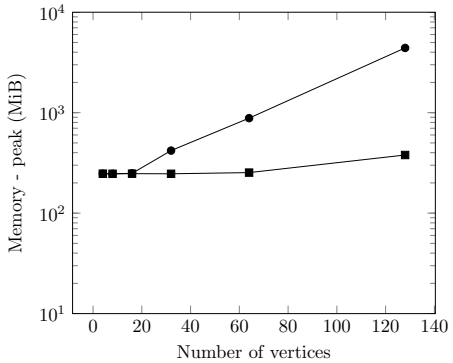
The last two implementations only compute the shortest path from a source to a target. A synchronization point is introduced after each main iteration (there are at most n synchronization points). These points slightly slow down the execution but reduce the memory usage (see Figures 6.2 and 6.3). The algorithms stop as soon as the target is scanned. In one implementation, both the source and the target are known. The algorithm outputs one of the shortest path along with its distance. In the other implementation, the source and the target are shared between players. They only learn the shortest distance between these two vertices. This approach may be useful as a subroutine for other algorithms. For example, the Chinese postman problem makes use of Dijkstra's shortest path to determine which edges have to be visited twice. It computes all the shortest paths between the odd degree vertices. If we wish to keep the degree of vertices secret, we may use such an implementation of Dijkstra. These last two implementations are feasible for a complete graph up to 256 vertices.



(a) Time in function of the number of vertices

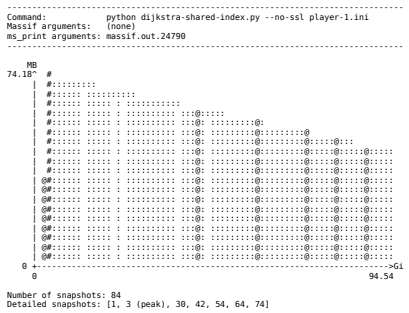


(b) Memory (rss) in function of the number of vertices

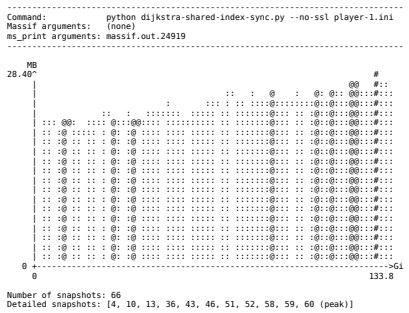


(c) Memory (peak) in function of the number of vertices

Figure 6.2: Time and memory usage (with the python memory-track argument) for the computation of the shortest path tree.



(a) without any synchronization



(b) with synchronization points

Figure 6.3: Memory usage (with valgrind) for the complete shortest path in a 16-vertex graph.

6.2.5 Leakage by Execution Flow: an Illustration

In this section, we show how the execution flow can reveal sensitive information even with a secure algorithm. We will run a standard Dijkstra's algorithm on the graph in Figure 6.4. It is an 8-vertex sparse graph. We want to compute the shortest-path tree rooted at vertex 0 and keep the source secret as well as the cost of the edges.

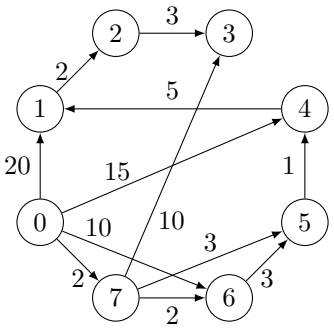


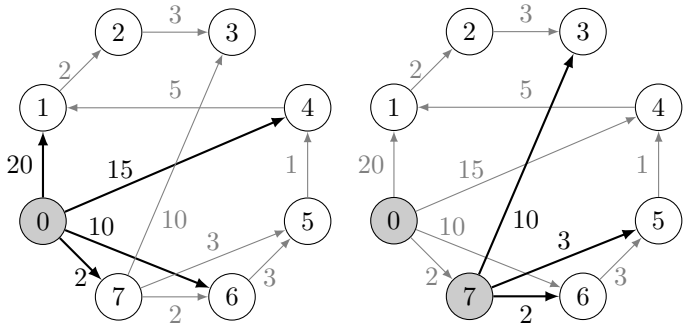
Figure 6.4: An 8-vertex graph.

Figure 6.5 shows the two first iterations of Dijkstra's classical algorithm. Once scanned the vertices are depicted in grey. During the first iteration, 4 neighbour vertices are explored and labelled (vertices 1, 4, 6 and 7). As we can see on Figure 6.4, vertex 0 is the only vertex to have 4 neighbours. It means that even with a source shared between the players, the execution flow reveals the source to all parties. The same issue can be observed on the second iteration. Vertex 7 is the only vertex that has 3 neighbours.

In the case of shortest path, the execution flow reveals at worst the order in which the vertices are scanned. This information does not reveal the actual distances but reveals, for example, that vertex 7 is the closest to the source. Depending on the application, this information leakage could remove the interest of dealing with secret-shared information. This is why we use a complete graph.

6.2.6 Secure Shortest Path with a Priority Queue

In this section, we use Toft's priority queue [52] in the secure shortest path based on Dijkstra's algorithm proposed in Section 6.2.2 (Protocol 9). Toft's priority queue is based on MPC primitives and contrary to ORAM implementations (e.g. Keller and Scholl [115]), it is deterministic. Toft's priority queue offers two operations: `PQ-insert`(p, x) which inserts a shared variable x with



(a) First iteration (b) Second iteration

Figure 6.5: The two first itration of Dijkstra's classical algorithm.

shared priority p in an initially empty list and `PQ-getmin`() which returns and removes from the list the pair (p, x) with the lowest priority p. The complexity of these two operations is $\mathcal{O}(\log^2(n))$ amortized (with n the overall number of operations performed). However, the priority queue does not support the `decrease-key`(p', x) operation which is of central interest for Dijkstra's algorithm. The role of `decrease-key` is to update (here decrease) the priority p of a variable x in the priority queue.

To avoid the `decrease-key` operation, Chen *et al.* proposed, broadly speaking, to replace it by a `PQ-insert` operation [122]. It implies outdated values in the priority queue: the same vertex may appear at multiple places with different priorities. The lowest priority is the updated one. After each `PQ-getmin` operation, it is necessary to test the status of the vertex to know if it is worth exploring or not (an outdated vertex will not improve any distance). Chen *et al.* do not consider the case of secure graphs but in the "clear" world, they claim that this kind of priority queue implementations make Dijkstra's algorithm even more efficient in practice. Indeed, Dijkstra's algorithm requires more `PQ-insert` and `PQ-getmin` operations than it would require from a priority queue supporting the `decrease-key` operation but the complexity of these `PQ-insert` and `PQ-getmin` operations is less. To support the `decrease-key` operation, priority queues use mechanisms that increase the complexity of the `PQ-insert` and `PQ-getmin` operations.

Algorithm. As usual, we have a graph $G = (V, E)$ where V is the list of vertices and E the list of edges, a matrix of shared weights [**W**], a vector of the distances from the source [**D**] and a source vertex [**s**]. The source vertex is given in a unary notation. The shared identity matrix [**U**] of size $|V|$ is used to insert

vertices represented as indices in a unary notation.

An element in the priority queue may have three different status. First, an element can be up-to-date, that is the only element that would appear in a priority queue with a `decrease-key` operation. Second, an up-to-date element (p, x) can become outdated if we perform a `PQ-insert`(p', x) operation. The vertex x has now a weaker priority p' and the element (p, x) becomes outdated. Finally, some elements are fake (with priority $\top$) and remain fake during the whole execution. To simplify the presentation we follow Keller and Scholl and do not track the status of the predecessor vector.

Protocol 10: SSSP3 protocol based on Dijkstra's algorithm with Toft's priority queue.

Input: A graph $G = (V, E)$ where V is the list of vertices and E the list of edges, a matrix of shared weights $[\mathbf{W}]_{i,j}$ for $i, j \in \{1, ..., |V|\}$, a shared identity matrix $[\mathbf{U}]_{i,j}$ and a source vertex $[\mathbf{s}] \in V$.

Output: The vector of distances $[\mathbf{D}]_i$ for $i \in \{1, ..., |V|\}$.

```
for i ← 1 to |V| do
    [D](i) ← [⊤];
end
updatevector([D], [s], [0]);
PQinsert([0], [s]);
for j ← 1 to |V| · (|V| − 1)/2 + 1 do
    [p], [x] ← PQgetmin();
    for i ← 1 to |V| do
        [a] ← ([D] + [W](∗, i)) · [x];
        [c] ← [a] < [D](i);
        [D](i) ← [D](i) + [c] · ([a] − [D](i));
        PQinsert([c] · [D](i) + (1 − [c]) · [⊤], [U](∗, i));
    end
end
return [D];
```

Analysis. The main loop of Protocol 10 is iterated $|V| \cdot (|V| - 1)/2 + 1$ times. It corresponds to the worst case scenario, where each remaining vertex is updated at each iteration with a lower priority. This means that we do not need to reveal the total number of updates performed on vertices during the algorithm. Moreover, we do not reveal neither which vertices have been updated, nor how many times they have been updated, nor when they have been updated. At no time in the execution, we reveal if the secret vertex we are treating is outdated or not. Thus, we do not stop the execution when coming across an outdated vertex. It avoids leaking potential information but increases the complexity.

This change does not compromise the correctness because an outdated vertex will not be able to improve any distance. We use also fake `PQ-insert` operations to standardize the execution flow.

The algorithm performs $\mathcal{O}(|V|^2)$ `PQ-getmin` operations and $\mathcal{O}(|V|^3)$ `PQ-insert` operations. The overall complexity is $\mathcal{O}(|V|^3)$ invocations of the comparison protocol and $\mathcal{O}(|V|^4)$ invocations of the multiplication protocol. In conclusion, it seems difficult to use efficiently Toft's priority queue in our secure setting. If we do not allow any information leakage, our algorithm has a tremendous overhead and is outperformed by our more basic implementation in Section 6.2.2.

6.3 Secure Maximum Flow

In an oriented graph where the edges have a constraint of *capacity*, the maximum flow problem consists in finding the maximum number of units that can be carried from a vertex called *source* to another vertex called *sink*. The *flow* through an edge designates the number of units passing by it. This number cannot exceed the capacity.

The first maximum flow algorithm was presented by Ford and Fulkerson in 1956 [123], [124]. Other solutions followed, for example, Edmonds-Karp algorithm [125], [126] and the Push-Relabel algorithm of Goldberg and Tarjan [127].

The maximum flow problem has numerous classical applications. In the spirit of our previous examples, one of them could be competing transport companies willing to determine the capacity they could reach if they decided to make a joint-venture. It is natural in such a context to expect that these companies will not be willing to disclose their full network structure to each other. As in the case of the shortest path, algorithms solving the maximum flow problem are also very useful as subroutines for solving other problems. The minimum cut problem is one such traditional example, which can be solved using $\mathcal{O}(|V|)$ invocations of the maximum flow algorithm. Solving this problem is then useful to determine where the weak points of the joint network would be.

In this section, we present two secure protocols for computing a maximum flow. The first one is based on Edmonds-Karp's algorithm and the second one is based on the Push-Relabel algorithm.

6.3.1 Edmonds-Karp's Algorithm

The basic idea of Edmonds-Karp's algorithm is to find an augmenting path in the residual graph that is the graph in which the edges are weighted by their residual capacity, i.e., the capacity minus the current flow. Each augmenting path increases the total flow so that the algorithm eventually terminates when there is no augmenting path left. The increase is monotonic and paths are considered once only. Typically, Edmonds-Karp's algorithm uses a breadth-first search to find the next augmenting path.

The asymptotic complexity of the traditional algorithm is $\mathcal{O}(|V||E|^2)$. As we have seen in the case of the shortest path problem, this complexity will be prohibitive even for very small graphs if they are complete. It therefore makes sense to focus our attention on (oriented) strongly sparse graphs, of which we consider the structure to be public. More precisely, we consider graphs in which the number of paths from the source to the sink is fairly small, e.g., bounded by a small polynomial in the number of vertices.

Protocol 11: SSMF1 maximum flow protocol based on Edmonds-Karp's algorithm.

Input: A graph $G = (V, E)$ where V is the list of vertices and E the list of edges, a source vertex $so \in V$, a sink vertex $si \in V$, and a list $\mathbf{p}$ of length k containing the paths between so and si sorted in a growing order of length. A set of capacities $[\mathbf{C}]_e$ for $e \in E$ and a set of flows $[\mathbf{F}]_e$ initially set to $[0]$ for $e \in E$. Edge $\bar{e}$ is the edge in the opposite direction of edge e.

Output: The maximum flow value from so to si.

while $|\mathbf{p}| > 0$ **do**
  $p \leftarrow \texttt{pop}(\mathbf{p})$;
  $[r], [\mathbf{i}] \leftarrow \texttt{binarymin}([\mathbf{C}(e)] - [\mathbf{F}(e)] \,|\, e \in p)$;
  $[b] \leftarrow [r] > 0$;
  $[a] \leftarrow [b] \cdot [r]$;
  for $e \in p$ **do**
    $[\mathbf{F}(e)] \leftarrow [\mathbf{F}(e)] + [a]$;
    $[\mathbf{F}(\bar{e})] \leftarrow [\mathbf{F}(\bar{e})] - [a]$;
  end
end
return $\sum_{e \in S} [\mathbf{F}(e)]$ where $S = \{e \in E | h(e) = so\}$;

The algorithm is given on input a list containing all the paths sorted in a growing order of length, $\mathbf{p} = (p_1, ..., p_k)$ where k is the number of paths in the graph. This list is not secret as the structure is not, and can therefore be

Number of paths	2	4	8	14	37	86	135
Number of edges	22	21	25	25	32	30	30
Execution times (sec)	3	6	9	18	40	94	148

Table 6.4: Execution times of Protocol 11 for 10-vertex graphs.

easily constructed in public. Our protocol based on Edmonds-Karp (the SSMF1 protocol) is presented in Protocol 11.

The main differences between this protocol and Edmonds-Karp's approach are:

- Our protocol uses a public enumeration of all the paths instead of a breadth-first search for capacity augmenting paths.
- Our protocol treats all the paths as if they were augmenting.

Protocol 11 is correct as the set of all the augmenting paths is contained in the set of all the paths $\mathbf{p}$. Moreover, it ensures the confidentiality of the edge capacities as no information is leaked about which path of $\mathbf{p}$ is augmenting and which is not.

As the length of the longest path in the graph is bounded by $|V|-1$, Protocol 11 requires $\mathcal{O}(k|V|)$ comparisons and $\mathcal{O}(k)$ multiplications. This protocol makes a crucial use of the existence of a small number of paths in the graph, something that we were not able to use in Protocol 9 for instance. It is however highly inefficient for dense graph and would have a factorial complexity for complete graphs.

This protocol applies well to our previous example of the three competing logistic companies trying to determine the max flow in their joint networks. If we consider a case with 10 vertices and 37 different paths, the execution takes less than a minute as shown in Table 6.4.

6.3.2 Push-Relabel Algorithm

The Push-Relabel algorithm, also called relabel-to-front when implemented with a FIFO list, introduces two additional attributes for the vertices, the height and the excess. An edge is called admissible if it goes from a higher to a lower vertex. The algorithm alternatively pushes the excess along admissible edges and increases the height of the vertices until all excess has been pushed to the sink or back to the source.

The basic operation of the algorithm is Push/Relabel applied to a given vertex. This operation pushes all the excess through incident admissible edges (updating the excesses of incident vertices accordingly). Finally, in case not all the excess has been pushed, the elevation of the vertex is minimally increased so as to create at least one more admissible edge, and Push/Relabel terminates.

Throughout the algorithm a list L with vertices with positive excess (except the source and the sink) is maintained. At each iteration, one vertex of L is selected and Push/Relabel is applied. The algorithm terminates when the list is empty. In the FIFO implementation, the next vertex of L to be treated is selected in the FIFO order. This FIFO Push/Relabel algorithm terminates in $\mathcal{O}(|V|^3)$ operations.

Our protocol based on Push-Relabel is presented in Protocol 12. The main differences between this protocol and the traditional Push/Relabel algorithm are as follows:

- When Push/Relabel is applied to a vertex with zero excess, no update of the elevation is performed at the end.
- In each phase, treat *all* vertices except the source and the sink, in a fixed order agreed between the players.
- During each Push/Relabel operation applied to a vertex i, the order in which the edges (i, j) are considered is fixed and agreed in advance between the players.

These changes do not modify the correctness of the original algorithm.

Moreover, it can be verified that the relabel-to-front algorithm terminates in maximum $4|V|^2 - 10|V| + 12$ complete phases. Therefore we obtain an "all-cases" complexity of $\mathcal{O}(|V|^2|E|)$, both in comparisons and multiplications. Note that this does not match the FIFO complexity, because we scan all edges at each pass, even when the excess of the tail vertex is zero.

The complexity of this algorithm remains lower than the one of the original Edmonds-Karp and it is asymptotically better than the optimized version of Edmonds-Karp presented in Section 6.3.1 for graphs with vertices of high degree. However, the running time of Protocol SSMF2 remains very high. Experiments showed that the use of a traditional halting criterion at the end of each SSMF2 phase (i.e. nothing has been pushed) results in dramatic running time improvements. However it also demonstrated a huge variability (the algorithm may halt after a single phase), which suggests that a substantial amount of information could be derived from it. Quantifying this information is left for future work, and its impact is likely to depend on the application.

Protocol 12: A phase of the SSMF2 protocol based on Push/Relabel.

Input: A complete graph $G = (V, E)$ where V is the list of vertices and E the list of edges. A vertex i to be treated, a vector of elevations $[\mathbf{H}]$, a matrix of residual capacities $[\mathbf{R}]$ and a vector of excesses $[\mathbf{Z}]$.

Output: Update of the elevations $[\mathbf{H}]$, the residual capacities $[\mathbf{R}]$ and the excesses $[\mathbf{Z}]$ for a phase.

$[\delta] \leftarrow 2|V|$;
for $j \leftarrow 1$ **to** $|E|$ **do**
  $[\alpha] \leftarrow [\mathbf{H}(i)] > [\mathbf{H}(j)]$;
  $[x] \leftarrow \min([\mathbf{Z}(i)], [\mathbf{R}(i, j)])$;
  $[y] \leftarrow [\alpha] \cdot [x]$;
  $[\mathbf{R}(i, j)] \leftarrow [\mathbf{R}(i, j)] - [y]$;
  $[\mathbf{R}(j, i)] \leftarrow [\mathbf{R}(j, i)] + [y]$;
  $[\mathbf{Z}(i)] \leftarrow [\mathbf{Z}(i)] - [y]$;
  $[\mathbf{Z}(j)] \leftarrow [\mathbf{Z}(j)] + [y]$;
  $[\delta] \leftarrow \min([\delta], [\mathbf{H}(j)] + 2|V| \cdot [\alpha])$;
end
$[\alpha] \leftarrow [\mathbf{Z}(i)] > 0$;
$[\mathbf{H}(i)] \leftarrow [\mathbf{H}(i)] \cdot (1 - [\alpha]) + ([\delta] + 1) \cdot [\alpha]$;

6.4 Conclusion

In this chapter, we proposed two protocols for securely computing shortest paths as well as two protocols for securely computing maximum flows in graphs. Besides the interest that these protocols have in the numerous contexts in which their insecure counterparts found applications in the past (possibly relying on a trusted third party), our investigation raised interesting complexity gaps between centralized algorithms and secure protocols, ranging from a constant to something growing like the number of vertices in the graphs. It is then natural to wonder whether these gaps, when they arise, can be decreased. Various avenues appear for that purpose:

- Design efficient data-structures adapted to the investigated problems. In particular, whether data structures similar to dynamic trees or Fibonacci heaps are implementable in a secured setting without revealing the execution flow remains an open question.

- Investigate whether secure comparisons, which often are a bottleneck, can be traded for other, cheaper, arithmetic operations. This raises unusual questions from a traditional algorithmic point of view, as comparisons are usually considered as basic operations.

Considering other standard combinatorial problems could also provide new insights. The protocols and results presented in the chapter are prototypes that validate the theoretical complexity evaluations. While the running times given for the protocols look impractical for large graphs, this issue must be put in perspective. Indeed, an implementation for concrete applications should definitively be improved by relying on lower level programming languages and optimized underlying libraries. Various optimization techniques (see, e.g., Bendlin *et al.* [53] or Damgård *et al.* [55]) would lead to performance increases of several orders of magnitude, as has been observed in the case of the AES during the last 3 years for instance (see, e.g. Damgård *et al.* [54] and the references within).

Part III

Conclusion

Chapter 7

Conclusions and Open Problems

Secure multi-party computation has been at the centre of cryptography research for almost 30 years. First, a series of foundational works demonstrated the possibility to evaluate any function in various models, the function being described as a circuit. Nowadays, research on this topic largely focuses on building practical solutions for specific problems. A lot of work has been carried out, for example, on benchmarking, auctions and voting applications.

One common point of these applications is that the function evaluation process is naturally oblivious of the inputs on which the function is evaluated. However, there are large classes of problems for which the natural evaluation process depends on the input data. In that case, even if all manipulated data are shared, the execution flow might leak undesirable information.

This thesis aims to address these complex problems that are not easy to represent in a secure setting. The originality of the thesis lies in the fact that it proposes the first secure protocols to solve some classical problems of game and graph theory. A second value added by this thesis lies in the fact that it analyses the various complexity gaps between the new secure algorithms and their traditional counterparts. These issues were largely ignored in previous work.

7.1 Contributions

Our first target was to design secure sorting algorithms. We proposed a sorting algorithm based on the Odd-even merge sorting network. This approach leads to an overhead by a constant factor. The asymptotic complexities are the same as the traditional ones. Then, we presented new sorting algorithms based on a unary representation of integers. These algorithms come with a linear overhead compared with their theoretical non-secure counterpart. This second approach is useful as a subroutine for an application using the same unary representation for integers. It avoids the extra costs linked with the change of representation.

Then, we studied the game-theoretic problem of fair division. More precisely, we studied the so-called cake-cutting problem that is the problem of dividing a heterogeneous good among parties with different interests. We developed a new secure procedure that does not have a game-theoretic equivalent. Indeed, we used the fact that utilities can be kept secret while performing computations on them. If we compare our oblivious protocol with its hypothetical non-oblivious counterpart, there is no overhead in term of asymptotic complexity. However, oblivious operations like comparisons are still more costly than their non-oblivious counterparts.

Finally, we studied different graph problems. Our work offers the first solutions for the secure evaluation of various graph properties. We compared in details different approaches that raise interesting complexity issues. For the single source shortest path, we compared two prototypes. The first one is based on Bellman-Ford's algorithm and the second one on Dijkstra's algorithm. Bellman-Ford is traditionally less efficient than Dijkstra. This is no longer true for the asymptotic complexities of our secure variants. Bellman-Ford is able to exploit the sparsity of the graph while Dijkstra always performs a shortest path on a complete graph. For the maximum flow, we compared two approaches based on Edmonds-Karp's algorithm and the Push-Relabel algorithm.

Our contributions illustrate a wide range of behaviours when passing to secure versions. There is no asymptotic overhead for sorting networks while there is a linear overhead for unary sorting. For the fair division problem, the most efficient way we came up with is to design a new procedure that does not have a counterpart in game-theory. Our new secure procedure enables reaching an equilibrium that dominates those that were previously known in unmediated procedures. For the graph problems, we obtained various overheads depending on the structure of the graph and on the algorithm used.

7.2 Perspectives

Several elements deserve future work following on from this thesis:

1. In this work, we identified several complexity gaps between traditional and secure approaches. These problems are little known in the current literature and raise several open questions. It would be interesting to better understand the source of these gaps and, for example, prove lower bounds or establish classes of problems with the same overhead. Problems with a pseudo-polynomial time algorithm seem worth of further exploration in this respect. As seen with the knapsack problem, these problems could allow using a unary representation without asymptotic overhead.

2. In the present work, we provide algorithms without any information leakage. However, this approach comes with important extra costs while leakage of some variables or elements of the structure could have no impact for some applications. An interesting future direction would be to study these security aspects in more details in order to precisely quantify the information leakage and its consequences. Leaking information could lead to real improvements in complexity. For example, when we compute a point-to-point shortest path, we can stop the execution once we have determined the shortest path. This would, however, reveal the number of iterations executed, but there might be contexts or types of graphs where this is harmless.

3. There are other classes of algorithms we could study, for example, sub-linear algorithms. These algorithms solve problems approximately by only exploring a small portion of them. They can be used to test whether a graph is bipartite or has a clique of a given size. Sub-linear algorithms estimate if the graph is far from satisfying the property or if it satisfies it. The error rate can be reduced by executing the property test many times. These algorithms are random and iterative which could be helpful to perform parallel executions. They could also lead to efficient solutions because they could easily use the technique of randomizing inputs, for example in graph problems, by working on an isomorphism of the graph.

4. Our oblivious algorithms have deterministic execution patterns. However, oblivious memories achieve obliviousness thanks to a probabilistic behaviour. They allow outsourcing some data on a remote server and accessing them in a way that makes it infeasible to know what specific piece of data has been accessed. Oblivious RAM might speed up secure multi-party computation when they involve large data. However, it comes with

a non-negligible computational overhead, $\mathcal{O}(\log^2 n)$, in the best solutions. Oblivious memories might even help for branching when the patterns of the execution of the branches are all identical. Oblivious RAM or similar oblivious data structures that can work without ORAM [52] could be a very useful tool in specific cases, as a complement of the techniques we use.

5. In this thesis, we used a Python library (VIFF) to prototype our algorithms. This framework was ideal to benchmark different implementations. However, the library in VIFF is quite limited. For instance, numerous algorithms have been proposed to speed up the online phase of MPC thanks to pre-computation, which is not done in VIFF. The complexity gaps might be different if we use pre-computation and focus on the complexity of the online computation phase only.

Appendix A

MPC Primitives

A.1 Protocols for secure comparison

In this section, we present a protocol for secure comparison introduced by Damgård *et al.* [57]. The bit decomposition protocol is the key tool to compare two shared secrets securely. A protocol for secure bit decomposition was also presented by Algesheimer *et al.* [86]. However, it is only passively secure and is not a constant round protocol. This section focuses on the bit decomposition protocol of Damgård *et al.*, which is the first constant round comparison protocol and the basis of the protocol used in the VIFF-based implementations.

Bit decomposition: $\texttt{BITS}([a]_p)$

Let p be a prime, $\log_2 p = \Theta(l)$. All modular arithmetic is done on $\mathbb{Z}_p = \{x \in \mathbb{Z} \mid 0 \leq x \leq p-1\}$, $p > k$ and p a prime number. Protocol 13 computes the bit-decomposition $a_0, \ldots, a_{l-1} \in \{0,1\}$ of $a = \sum_{i=0}^{l-1} a_i 2^i$ with $a \in \mathbb{F}_p$.

The first idea of the protocol for computing the bit-decomposition of a shared value $[a]_p$ is to use the bit-decomposition of a random value $[b]_p$. This idea is in line with the idea used for the inversion of a shared element. The shared value is "hidden" by a random value $[c]_p \leftarrow [a]_p - [b]_p$. This new value $[c]_p$ can then be revealed without leaking information about the shared value and used for further computation. This is done in Lines 1 to 4 of Protocol 13.

As explained by Damgård [57], $c = a - b \mod p$ and $d = c + b$ (in the integers). Therefore, $d = a + pq$ for some $q \in \{0,1\}$. Since $a \in \{0, \ldots, p-1\}$, it follows

Protocol 13: $[a]_B \leftarrow \texttt{BITS}([a]_p)$

1 The input is $[a]_p$, where $a \in \mathbb{F}_p$.
2 $([b_0]_p, \ldots, [b_{l-1}]_p, [b]_p) \leftarrow \texttt{SOLVED-BITS}()$.
3 $[a-b]_p \leftarrow [a]_p - [b]_p$.
4 $c \leftarrow \texttt{REVEAL}([a-b]_p)$, where $c \in \mathbb{F}_p$.
5 $[d]_B \leftarrow \texttt{BIT-ADD}(c, [b]_B)$, where $[d]_B = ([d_0]_p, \ldots, [d_l]_p)$.
6 $[q]_p \leftarrow \texttt{BIT-LT}(p, [d]_B)$.
7 $(f_0, \ldots, f_{l-1}) = \texttt{BITS}(2^l - p)$, the bitwise representation of the positive integer $2^l - p$.
8 For $i = 0, \ldots, l-1$ in parallel: $[g_i]_p = f_i[q]_p$.
9 $[g]_B = ([g_0]_p, \ldots, [g_{l-1}]_p)$.
10 $[h]_B \leftarrow \texttt{BIT-ADD}([d]_B, [g]_B)$, where $[h]_B = ([h_0]_p, \ldots, [h_{l+1}]_p)$.
11 $[a]_B = ([h_0]_p, \ldots, [h_{l-1}]_p)$.
12 Output $[a]_B$.

that $q = 1$ iff $p < d$. A sharing of q is computed in Line 6.

Then, the following equalities hold: $f = 2^l - p$ (Line 7), $g = qf = q2^l - qp$ in the integers (Line 8), $h = d + g = (a + qp) + (q2^l - qp) = a + q2^l$ (Line 10). Now h is either a or $a + 2^l$. The problem is solved by computing $h \mod 2^l$, i.e., dropping the two most significant bits of h since $[h]_B = ([h_0]_p, \ldots, [h_{l+1}]_p)$. This is the key idea of this protocol.

This protocol leaks no information as long as all sub-protocols are private. The value c, revealed in Line 4, is uniformly random in $\mathbb{F}_p$ and leaks thus no information about a. The total complexity is 114 rounds and $110l \log_2 l + 118l$ invocations of the multiplication protocol.

Random solved bits: SOLVED-BITS()

The protocol $([b]_B, [b]_p) \leftarrow \texttt{SOLVED-BITS}()$ has no inputs. It outputs shares of a uniformly random element $b \in \mathbb{F}_p$ and its bit decomposition $(b_0, \ldots, b_{l-1})$.

The protocol $\texttt{RAN}_2()$ securely generates for $i = 0, \ldots, l-1$ a sharing $[b_i]_p$ of a uniformly random bit $b_i \in \{0, 1\} \subseteq \mathbb{F}_p$ (Line 1). Then, $[b]_B$ is by construction the bit-wise sharing of $[b]_p$ (Line 2) and b is uniformly random from $\left\{0, 1, \ldots, 2^{l-1}\right\}$. So under the condition that the protocol does not abort (Line 5), b is uniformly random from $\{0, 1, \ldots, p-1\}$.

The choice of the prime p is very important for this protocol. If $b \geq p$, it does not fulfill the output requirement, i.e., $b \in \mathbb{Z}_p$. The prime p has to be chosen in such a way that the protocol only aborts with a small probability, i.e.,

Protocol 14: $([b]_B, [b]_p) \leftarrow$ `SOLVED-BITS`()

1 For $i = 0, \ldots, l-1$ in parallel: $[b_i]_p \leftarrow \texttt{RAN}_2()$.
2 $[b]_B = ([b_0]_p, \ldots, [b_{l-1}]_p)$.
3 $[c]_p \leftarrow \texttt{BIT-LT}([b]_B, p)$.
4 $c \leftarrow \texttt{REVEAL}([c]_p)$.
5 If $c = 0$, then abort. Otherwise proceed as below.
6 $[b]_p \leftarrow \sum_{i=0}^{l-1} 2^i [b_i]_p$.
7 Output $([b]_B, [b]_p)$.

$b \geq p$ with a small probability. When the protocol does not abort, it leaks no information except for Line 4: the protocol reveals that $b < p$. However, this is not an "information" since $b \in \mathbb{Z}_p$. The total complexity is 21 rounds and $96l$ invocations of the multiplication protocol.

Bitwise Less-Than: `BIT-LT`$([a]_B, [b]_B)$

The protocol $([c]_p) \leftarrow \texttt{BIT-LT}([a]_B, [b]_B)$ computes a sharing of the bit $(a \overset{?}{<} b) \in \{0, 1\}$, where $(a \overset{?}{<} b) = 1$ iff $a < b$.

Protocol 15: $([c]_p) \leftarrow \texttt{BIT-LT}([a]_B, [b]_B)$

1 For $i = 0, \ldots, l-1 : [e_i]_p \leftarrow \texttt{XOR}([a_i]_p, [b_i]_p)$.
2 $([f_{l-1}]_p, \ldots, [f_0]_p) = \texttt{PRE}_\vee([e_{l-1}]_p, \ldots, [e_0]_p)$.
3 $[g_{l-1}]_p = [f_{l-1}]_p$.
4 For $i = 0, \ldots, l-2 : [g_i]_p \leftarrow [f_i]_p - [f_{i+1}]_p$.
5 For $i = 0, \ldots, l-1 : [h_i]_p \leftarrow \texttt{MUL}([g_i]_p, [b_i]_p)$.
6 $[h]_p \leftarrow \sum_{i=0}^{l-1} [h_i]_p$.
7 Output $[h]_p$.

In Line 1, the protocol $[e_i]_p \leftarrow \texttt{XOR}([a_i]_p, [b_i]_p)$ is computed in one round: the local computation $[d]_p \leftarrow [a_i]_q - [b_i]_q$ is followed by the protocol $[e]_p \leftarrow \texttt{MULT}([d]_p, [d]_p)$. In Line 2, the protocol $([f_{l-1}]_p, \ldots, [f_0]_p) = \texttt{PRE}_\vee$ $([e_{l-1}]_p, \ldots,$ $[e_0]_p)$ computes the prefix-or $([f_{l-1}]_p, \ldots, [f_0]_p)$, where $f_i = \vee_{j=i}^{l-1} e_j$. Let assume that $a \neq b$, and let i_0 denote the largest index i, where $a_i \neq b_i$. Then $a < b$ iff $b_{i_0} = 1$. Note that i_0 is the largest i for which $f_i = 1$, and thus $g_i = 1$ iff $i = i_0$. Therefore, $h = b_{i_0}$. In the special case $a = b$, clearly $h = 0$, as it should be.

The idea behind this protocol can easily be seen with the following example where $b > a$:

i	$l-1$	$l-2$	...	i_0+1	i_0	i_0-1	...	2	1	0
a	0	1	...	1	0	0	...	1	0	0
b	0	1	...	1	1	1	...	1	1	0
e	0	0	...	0	1	1	...	0	1	0
f	0	0	...	0	1	1	...	1	1	1
g	0	0	...	0	1	0	...	0	0	0
h	0	0	...	0	1	0	...	0	0	0

Here, $h_{i_0} = 1$ because $b > a$. The protocol is private because only private sub-protocols are called. The total complexity is 19 rounds and $22l$ invocations of the multiplication protocol.

A.2 Inversion of a polynomially shared element

Inversion of a polynomially shared element: $\texttt{INV}([a]_i^q)$ [128]. Let $\mathcal{Z}_q$ be the set $\{x \in \mathbb{Z} | -q/2 < x < q/2\}$.

Protocol 16: $[x]_i^q \leftarrow \texttt{INV}([a]_i^q)$

1 $[r]_i^q \leftarrow \texttt{JRP}(\mathcal{Z}_q)$
2 $[y]_i^q \leftarrow \texttt{MUL}([r]_i^q, [a]_i^q)$
3 $y \leftarrow \texttt{REVEAL}[y]_i^q$
4 If $y = 0$, then abort. Otherwise proceed as below.
5 $[x]_i^q \leftarrow \texttt{LOC-MUL}(y^{-1}, [r]_i^q)$

Line 1 (joint random sharing over $\mathcal{Z}_q$): Player P_i holds a share of the random value r. The goal of this protocol is to generate shares of a secret random element from $\mathcal{Z}_q$. Each player chooses a random number $r_j \in \mathcal{Z}_q$, shares it according to Shamir scheme and sends the shares to the respective players. Then each player adds up all the received shares rem q to obtain the share of a random value. The protocol requires $O(1)$ round and $O(lk^2 \log_2 k)$ bit operations per player.

Line 2 (multiplication): Player P_i has a share $[y]_i^q$ of the value $y = r \cdot a$.

Line 3 : Player P_i reveals $[y]_i^q$ without leaking any information about $[a]_i^q$ since $[r]_i^q$ is his secret share of the random value r.

Line 5 : Player P_i gets his share $[x]_i^q$ of $x = a^{-1}$ by computing $[x]_i^q = y^{-1} \cdot [r]_i^q$. Actually, he performs the following computation: $([r]_i^q \cdot a)^{-1}[r]_i^q = a^{-1}$.

The round complexity is $O(1)$. The protocol requires an expected number of $O(l^2k + lk^2 \log_2 k)$ bit operations per player.

Appendix B

Standard Definitions of Game Theory

A game is an interaction between rational players who have objectives and constraints. Each player has a set of possible actions. Each combination of these individual actions has a collective consequence, which has an individual outcome for each player. Each player has preferences on his outcome. The standard definitions introduced in this appendix are taken from Osborne [129],[130].

In the "game of chicken" (Aumann, 1974), two players drive on a single lane road in opposite directions. Both have two possible actions: either go on driving ("dare") or brake ("chicken out"). If both players go on driving, the collective consequence will be a head-on crash. The outcome for each player will be death, outcome for which he has his preferences. The collective consequence and the individual outcomes have to be distinguished. For example, in an asymmetrical game where one player drives a car and the other a truck, the collective consequence will still be a head-on crash but the individual outcome will be worse for the car driver.

Normal form game (Strategic game)

A k-player game $\Gamma = \left(\{A_i\}_{i=1}^{k}, \{u_i\}_{i=1}^{k}\right)$, presented in normal form, is determined by specifying, for each player P_i, a set of possible *actions* A_i and a *utility function* $u_i : A_1 \times \cdots \times A_k \mapsto \mathbb{R}$. Letting $A \stackrel{\text{def}}{=} A_1 \times \cdots \times A_k$, we refer to a tuple of actions $\boldsymbol{a} = (a_1, \ldots, a_k) \in A$ as an *outcome*. The utility function u_i of

party P_i expresses this player's preferences over outcomes: P_i prefers outcome $\boldsymbol{a}$ to outcome $\boldsymbol{a}'$ iff $u_i(\boldsymbol{a}) > u_i(\boldsymbol{a}')$.

A two-player strategic game can be represented in matrix form. Suppose that Alice and Bob take part to the "game of chicken". They both have the following (symmetric) preferences: the two drivers agree that the head-on crash is the worst way out: $(D, D) = (0, 0)$. The situation where they both give up and both feel ashamed, ranks second: $(C, C) = (4, 4)$. The best situation for Alice is when Bob gives up and she does not and vice-versa for Bob: $(D, C) = (5, 1)$ and $(C, D) = (1, 5)$. Table B.1 is the matrix form representation of this game. Alice's preferences are given in the first place (in black) while Bob's preferences are given in the second place (in grey). The best action for a party to play in response to each action of the opponent is underlined.

	Chicken out	Dare
Chicken out	(4 , 4)	($\underline{1}$, $\underline{5}$)
Dare	($\underline{5}$, $\underline{1}$)	(0 , 0)

Table B.1: Matrix form representation of the game of chicken.

Nash equilibria

Let $\Gamma = \left(\{A_i\}_{i=1}^k, \{u_i\}_{i=1}^k\right)$ be a game presented in normal form, and let $A = A_1 \times \cdots \times A_k$. A tuple $\boldsymbol{a} = (a_1, \ldots, a_k) \in A$ is a *pure strategy Nash equilibrium* if for any i and any $a_i' \in A_i$ it holds that $u_i(a_i', \boldsymbol{a}_{-i}) \leq u_i(\boldsymbol{a})$, where $\boldsymbol{a}_{-i} \stackrel{\text{def}}{=} (a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_k)$.

In a normal form game it is easy to find pure strategy Nash equilibria (if they exist). Every player identifies the best action(s) to play in response to each action of the other player. For example, if Bob chickens out, the best action for Alice is to dare (she has a payoff of 5 instead of 4 if she chickens out). If Bob dares, the best response for Alice is to chicken out. Best responses are underlined in Table B.1. A combination of best responses is a pure strategy Nash equilibrium.

The "game of chicken" has two pure strategy Nash equilibria: NE1=(C, D) with payoffs (1,5) and NE2=(D, C) with payoffs (5,1).

Let $\Gamma = \left(\{A_i\}_{i=1}^k, \{u_i\}_{i=1}^k\right)$ be as above, and let σ_i be a distribution over A_i. Then $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_k)$ is a *mixed strategy Nash equilibrium* if for any i and any

distribution σ_i' over A_i it holds that $u_i(\sigma_i, \boldsymbol{\sigma}_{-i}) \leq u_i(\boldsymbol{\sigma})$.

Assigning probabilities to tuples of actions corresponds to a widening of the set of accessible payoffs (pure strategies are particular mixed strategies). There is an additional mixed strategy equilibrium in the "game of chicken": NE3=($\frac{1}{2} \cdot D + \frac{1}{2} \cdot C, \frac{1}{2} \cdot D + \frac{1}{2} \cdot C$) with payoffs ($\frac{5}{2}, \frac{5}{2}$).

Nash equilibrium is an interesting notion since Nash's theorem states that any strategic game, in which the strategies are finite, has at least one mixed strategy equilibrium. In a Nash equilibrium, all players follow their strategy independently of each other; however, it is often possible to reach higher expected payoffs with correlated strategies. Achieving a correlated equilibrium requires the presence of a mediator.

Correlated equilibria

Let $\Gamma = (\{A_i\}, \{u_i\})$. Let $\Delta(A)$ denote the set of probability distributions over the finite set A. A distribution $\mathcal{M} \in \Delta(A)$ is a *correlated equilibrium* if for any $\boldsymbol{a} = (a_1, \ldots, a_k)$ in the support of $\mathcal{M}$, any i, and any $a_i' \in A_i$, it holds that $u_i(a_i', \boldsymbol{a}_{-i} | a_i) \leq u_i(\boldsymbol{a} | a_i)$. Note that $u_i(a_i', \boldsymbol{a}_{-i} | a_i)$ denotes the expected utility of P_i, given that he plays action a_i' after having received recommendation a_i and all other parties play their recommended actions $\boldsymbol{a}_{-i}$.

Roughly speaking, in a correlated equilibrium no player has an incentive to deviate from his recommended strategy, i.e., from the mediator's recommendation. For the game of chicken, the best correlated equilibrium is CE1=($\frac{1}{3}(C,D) + \frac{1}{3}(D,C) + \frac{1}{3}(C,C)$) with payoffs ($3\frac{1}{3}, 3\frac{1}{3}$) The payoffs of this correlated equilibrium are higher than the payoffs of the mixed strategy Nash equilibrium. Furthermore, the payoffs are symmetric. The notion of symmetric strategy profile seems in line with the notion of equitability. However, it is only an "expected" equitability: if there is only one game (one-shot game), a player wins 5, 4 or 1. The result is "equitable" in only one situation, when the mediator recommends (C,C)=(4,4).

Mediators can generally expand the set of equilibria in all directions, thus they do not necessarily increase the parties' payoffs. Mediators can also force the parties into worse payoffs than in the unmediated game: CE2=($\frac{1}{3}(C,D) + \frac{1}{3}(D,C) + \frac{1}{3}(D,D)$) with payoffs $(2,2)$ Implementing such a mediator is, however, not really interesting.

Correlated equilibria always exist and form a convex set which necessarily includes the convex hull of Nash equilibria. The highest payoffs achievable under

the game theoretic settings correspond to correlated equilibria. Contrary to an arbitrator, a mediator cannot force the players to follow his recommendation. The mediator has to give his recommendation in such a way that no player has an incentive to deviate from it. This restricts the set of payoffs achievable under the game theoretic settings.

Feasible payoff profile

The set of *feasible payoff profiles* of a strategic game is the set of all weighted averages of payoff profiles in the game.

In the game of chicken, an arbitrator would force both players to follow the careful strategy: chickening out (C, C) with payoffs (4,4). The combination of the careful strategies does not lead to an equilibrium. If Alice knows that Bob will chicken out, she will dare. This payoff is not an equilibrium and is thus not achievable under the game theoretic settings.

The notion of arbitrator does not fit to the game theoretic settings because it does not take into account the rationality of the players. No player will follow the recommendation of an arbitrator if it is not in his own interest. However, this notion seems in line with the cryptographic honest-but-curious model. In this model all players, even the corrupted ones, have to follow the protocol.

Set of expected payoffs in the game of chicken

All the payoffs inside the convex hull of the Nash equilibria correspond to achievable payoffs of correlated equilibria. The set of correlated equilibria can be larger than the convex hull of the Nash equilibria. This is the case for the "game of chicken".

The only two pure strategy Nash equilibria of the "game of chicken" are NE1=(C, D) with payoffs (1,5) and NE2=(D, C) with payoffs (5,1). The game has one more mixed strategy Nash equilibrium: NE3=$(\frac{1}{2} \cdot D + \frac{1}{2} \cdot C, \frac{1}{2} \cdot D + \frac{1}{2} \cdot C)$ with payoffs $(\frac{5}{2}, \frac{5}{2})$. The convex hull of the Nash equilibria is delimited by the triangle NE1-NE2-NE3 in Figure B.1.

The payoffs inside the convex hull of Nash equilibria are always achievable by a correlated equilibrium. However, in this game the set of correlated equilibria is even larger than the convex hull of Nash equilibria. Two possible correlated equilibria outside this convex hull are CE1=$(\frac{1}{3}(C, D) + \frac{1}{3}(D, C) + \frac{1}{3}(C, C))$ with

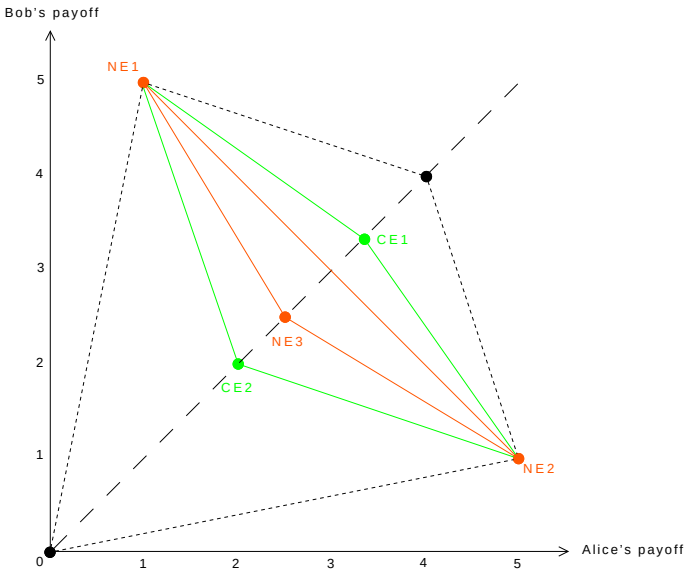


Figure B.1: Equilibria of the "game of chicken".

payoffs $(3\frac{1}{3}, 3\frac{1}{3})$ and CE2=$(\frac{1}{3}(C,D) + \frac{1}{3}(D,C) + \frac{1}{3}(D,D))$ with payoffs $(2,2)$. The set of correlated equilibria is delimited by the quadrilateral NE1-CE1-NE2-CE2 while the set of feasible payoff profiles is represented by the dotted lines in Figure B.1.

In term of achievable payoffs, the following inclusions hold for all games: {pure strategies} $\subset$ {mixed strategies} $\subset$ {correlated strategies}.

Appendix C

Cake-Cutting in the Game-Theoretic Setting

Appendix B showed how a mediator can improve the expected players' payoffs for a simple game. The cake-cutting is a more complex problem but the idea remains the same: a mediator can potentially improve the players' payoffs. However, it is not so easy to switch from simple games to the cake-cutting problem because players do not directly play a "cake-cutting game", they follow a procedure. This procedure, designed to produce a fair division, is the game.

Extensive form game

A game theoretic procedure proceeds by steps. The players do no longer act simultaneously: in each step, a set of players execute some actions. "A *strategy* for a player is now an adaptive sequence of moves consistent with the procedure, which the participants choose sequentially when called upon by the procedure" [87]. Such a situation is modelled by an extensive form game.

Extensive form games remove the assumption that players act simultaneously. Playing the game defines a *history* of the actions taken by the players thus far, and a player P_i's strategy σ_i now specifies, for each step in which it is P_i's turn to move, a (randomized) function mapping possible histories to actions. Players' utilities are now functions of terminal histories (i.e., histories that occur at the end of the game), rather than functions of the strategy vector of the players [92].

An example is given for the most simple cake-cutting procedure: "I cut, you choose".

"I cut, you choose" procedure

"I cut, you choose" is the easiest proportional procedure with two players. In the procedures quoted from Brams and Taylor [87], all strategic aspects are in parentheses and the arguments that the strategies perform are placed between steps and labeled as "Aside". Let Alice and Bob be the two players.

> Step 1. Alice cuts a rectangular cake into two pieces (that she considers to be the same size).
>
> Step 2. Bob chooses a piece (that he considers to be at least tied for largest).
>
> Aside. Clearly, Alice's strategy guarantees her a piece of size exactly 1/2 in her measure, while Bob's strategy guarantees him a piece of size at least 1/2 in his measure.

Suppose Alice and Bob want to divide a cake, half vanilla and half chocolate. Alice likes chocolate as well as vanilla and, therefore, cuts the cake exactly in the middle. The division gives two homogeneous parts: one of vanilla and one of chocolate. Bob is only interested in vanilla and perceives chocolate as insignificant. He thus chooses the vanilla part. According to his own subjective valuation, he receives all the value of the cake while Alice only gets half of it (in her own valuation). This allocation is not at all equitable: Bob values his piece twice as much as Alice does.

This procedure can be modelled as an extensive form game but it cannot be represented in a matrix form. Alice has an infinite set of possible actions: she can cut the cake wherever she wants. It is a game with an infinity of subgames: one for every possible cut. However, by restricting the set of possible actions, it can be modelled in a matrix form.

Suppose that Alice only has two possible actions: making a conservative division (division that is exactly 50-50 in her own estimation to garantee herself half of the cake) or making an exploitative division (divison that is almost 50-50 in Bob's estimation). Thus, in a conservative division, Alice cuts the cake exactly in the middle while in an exploitative division, she cuts the cake into a first part with half the vanilla plus a small slice ϵ of it and a second part with the remaining vanilla plus the whole chocolate. To achieve an exploitative division, Alice needs to know Bob's preferences. Bob still has only two possible actions:

taking the largest part or taking the smallest one. The idea of making an exploitative or conservative division is taken from Brams and Taylor [82]. This simplified game also illustrates the role of information and thus the importance of secrecy.

"I cut, you choose" game in extensive form

Let us take as a working hypothesis that Alice and Bob have complete information (it is common knowledge that Alice likes chocolate as well as vanilla and that Bob is only interested in vanilla). This situation can be modelled as an extensive game represented by the tree in Figure C.1. Let the small slice be $\epsilon = 1/100$ of the cake. The payoffs correspond to the value that the players give to the piece they have received.

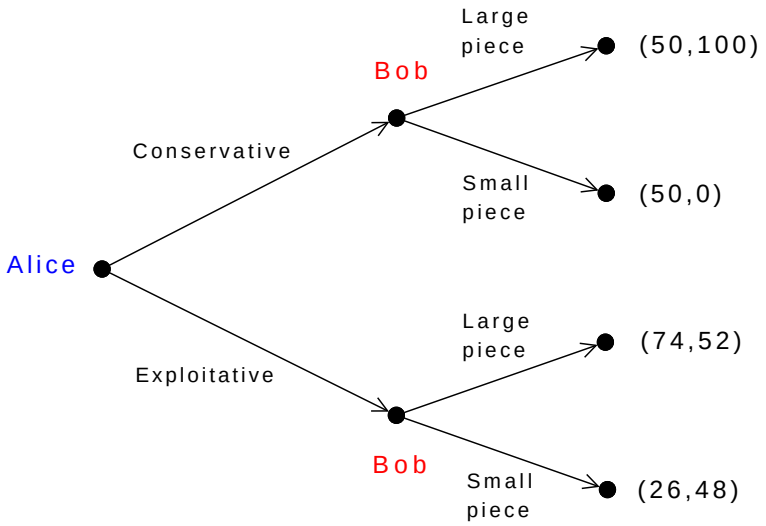


Figure C.1: "I cut, you choose" game.

This game is alternatively represented in a matrix form (Table C.1). Contrary to the normal form game, Bob has four different strategies. In an extensive form game, Bob has to specify an action for all possible actions of Alice. For example, Bob's strategy LS means that he takes the largest piece if Alice is conservative while he takes the smallest one if she is exploitative.

By modelling the game in this way, three pure strategy Nash equilibria can be seen. The most interesting is (C, LS). This equilibrium shows that if Alice

	LL	*LS*	*SL*	*SS*
C	(50 , 100)	(50 , 100)	(50 , 0)	(50 , 0)
E	(74 , 52)	(26 , 48)	(74 , 52)	(26 , 48)

Table C.1: Extensive form of "I cut, you choose".

knows that Bob hates being exploited, - so much that he will in this case take his revenge by choosing the "unrational" strategy, i.e., the smallest part -, she will be conservative.

A mediator could improve the payoffs by advising, for example, the following strategy: CE=$(\frac{1}{3}(C, LS) + \frac{2}{3}(E, LL))$ with almost symmetric expected payoffs $(66, 68)$. This is a correlated equilibrium: neither Alice, nor Bob has an incentive to deviate from its recommended strategy.

Implementing such a mediator thanks to MPC is implementing a probabilistic function. Such a mediator is useless to design an equitable one-shot procedure because the payoffs of a correlated equilibrium are only symmetric *expected* payoffs.

"I cut, you choose" game in normal form

Let us take as a working hypothesis that Alice has a complete information on Bob's preferences while Bob does not know Alice's preferences. This situation can be modelled as a normal form game (Table C.2. The game has a unique equilibrium: (E, L)=(74,52).

	L	*S*
C	(50 , 100)	(50 , 0)
E	(74 , 52)	(26 , 48)

Table C.2: Normal form of "I cut, you choose".

The "I cut, you choose" game illustrates the importance of secrecy. If Alice knows Bob's preferences and if Bob does not know hers, he will never receive more than 52. Note that Bob cannot detect if he is exploited because he does not know Alice's preferences.

Bibliography

[1] Mawet, S., Pereira, O., Petit, C.: Equitable cake cutting without mediator. In Nikova, S., Batina, L., eds.: 5th Benelux Workshop on Information and System Security. (2010)

[2] Aly, A., Cuvelier, E., Mawet, S., Pereira, O., Vyve, M.V.: Securely solving simple combinatorial graph problems. In: Financial Cryptography. Volume 7859 of Lecture Notes in Computer Science., Springer (2013) 239–257

[3] Yao, A.C.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, IEEE (1982) 160–164

[4] Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC, ACM (1987) 218–229

[5] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC, ACM (1988) 1–10

[6] Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: STOC, ACM (1988) 11–19

[7] Goldreich, O.: The Foundations of Cryptography - Volume 1, Basic Techniques. Cambridge University Press (2001)

[8] Goldreich, O.: The Foundations of Cryptography - Volume 2, Basic Applications. Cambridge University Press (2004)

[9] Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)

[10] Goldreich, O.: Secure multi-party computation. (2000)

[11] Cramer, R., Damgård, I.: Multiparty computation, an introduction. In: Contemporary Cryptology. Advanced Courses in Mathematics - CRM Barcelona. Birkhäuser Basel (2005) 41–87

[12] Fitzi, M., Hirt, M., Maurer, U.M.: General adversaries in unconditional multi-party computation. In: ASIACRYPT. Volume 1716 of Lecture Notes in Computer Science., Springer (1999) 232–246

[13] Yao, A.C.: How to generate and exchange secrets. In: FOCS, IEEE (1986) 162–167

[14] Shamir, A.: How to share a secret. Communications of the ACM **22**(11) (1979) 612–613

[15] Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In: Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, ACM (1998) 101–111

[16] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: FOCS, IEEE Computer Society (1995) 41–50

[17] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3) (1996) 431–473

[18] Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: STOC, ACM (1997) 294–303

[19] Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Proceedings of the 8th TCC, Springer-Verlag (2011) 144–163

[20] Bogetoft, P., Damgård, I., Jakobsen, T.P., Nielsen, K., Pagter, J., Toft, T.: A practical implementation of secure auctions based on multiparty integer computation. In: Financial Cryptography. Volume 4107 of LNCS., Springer (2006) 142–147

[21] Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T.P., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M.I., Toft, T.: Secure multiparty computation goes live. In: Financial Cryptography. Volume 5628 of Lecture Notes in Computer Science., Springer (2009) 325–343

[22] Miltersen, P.B., Nielsen, J.B., Triandopoulos, N.: Privacy-enhancing auctions using rational cryptography. In: Advances in Cryptology -

CRYPTO 2009, 29th Annual International Cryptology Conference. Volume 5677 of Lecture Notes in Computer Science., Springer (2009) 541–558

[23] Cramer, R., Franklin, M.K., Schoenmakers, B., Yung, M.: Multi-autority secret-ballot elections with linear work. In: EUROCRYPT. Volume 1070 of LNCS., Springer (1996) 72–83

[24] Clarkson, M.R., Chong, S., Myers, A.C.: Civitas: Toward a secure voting system. In: IEEE Symposium on Security and Privacy, IEEE (2008) 354–368

[25] Adida, B., de Marneffe, O., Pereira, O., Quisquater, J.: Electing a university president using open-audit voting: Analysis of real-world use of helios. In: Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '09, USENIX Association (2009)

[26] Gjøsteen, K.: Analysis of an internet voting protocol. IACR Cryptology ePrint Archive (2010) 380

[27] Gjøsteen, K.: The norwegian internet voting protocol. In: E-Voting and Identity - Third International Conference, VoteID 2011. Volume 7187 of Lecture Notes in Computer Science., Springer (2012) 1–18

[28] Barni, M., Failla, P., Kolesnikov, V., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Secure evaluation of private linear branching programs with medical applications. In: ESORICS. Volume 5789 of LNCS., Springer (2009) 424–439

[29] Lindell, Y., Pinkas, B.: Privacy preserving data mining. In: CRYPTO. Volume 1880 of Lecture Notes in Computer Science., Springer (2000) 36–54

[30] Lindell, Y., Pinkas, B.: Privacy preserving data mining. J. Cryptology **15**(3) (2002) 177–206

[31] Lindell, Y., Pinkas, B.: Secure multiparty computation for privacy-preserving data mining. IACR Cryptology ePrint Archive (2008) 197

[32] Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. Int. J. Inf. Sec. **11**(6) (2012) 403–418

[33] Bogdanov, D., Talviste, R., Willemson, J.: Deploying secure multi-party computation for financial data analysis - (short paper). In: Financial Cryptography and Data Security. Volume 7397 of Lecture Notes in Computer Science., Springer (2012) 57–64

[34] Kamm, L., Bogdanov, D., Laur, S., Vilo, J.: A new way to protect privacy in large-scale genome-wide association studies. Bioinformatics **29**(7) (2013) 886–893

[35] Bogdanov, D., Kamm, L., Laur, S., Pruulmann-Vengerfeldt, P., Talviste, R., Willemson, J.: Privacy-preserving statistical data analysis on federated databases. In: Privacy Technologies and Policy - Second Annual Privacy Forum. Volume 8450 of Lecture Notes in Computer Science., Springer (2014) 30–55

[36] Erkin, Z., Franz, M., Guajardo, J., Katzenbeisser, S., Lagendijk, I., Toft, T.: Privacy-preserving face recognition. In: Privacy Enhancing Technologies. Volume 5672 of Lecture Notes in Computer Science., Springer (2009) 235–253

[37] Sadeghi, A.R., Schneider, T., Wehrenberg, I.: Efficient privacy-preserving face recognition. In: ICISC. Volume 5984 of LNCS., Springer (2009) 229–244

[38] Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: ASIACRYPT. Volume 5912 of LNCS., Springer (2009) 250–267

[39] Shelat, A., Shen, C.H.: Two-output secure computation with malicious adversaries. IACR Cryptology ePrint Archive (2011) 533

[40] Nielsen, J.B., Nordholt, P.S., Orlandi, C., Burra, S.S.: A new approach to practical active-secure two-party computation. In: CRYPTO. Volume 7417 of Lecture Notes in Computer Science., Springer (2012) 681–700

[41] Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - secure two-party computation system. In: USENIX Security Symposium, USENIX (2004) 287–302

[42] Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: ACM Conference on Computer and Communications Security, ACM (2008) 257–266

[43] Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: ESORICS. Volume 5283 of Lecture Notes in Computer Science., Springer (2008) 192–206

[44] Burkhart, M., Strasser, M., Dimitropoulos, X.A.: SEPIA: Security through private information aggregation. CoRR **abs/0903.4258** (2009)

[45] Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.A.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: USENIX Security Symposium, USENIX Association (2010) 223–240

[46] Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: Tasty: tool for automating secure two-party computations. In: ACM Conference on Computer and Communications Security, ACM (2010) 451–462

[47] Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Public Key Cryptography - PKC 2009. Volume 5443 of Lecture Notes in Computer Science., Springer (2009) 160–179

[48] Geisler, M.: Cryptographic protocols: theory and implementation. PhD thesis, Aarhus University Denmark, Department of Computer Science (2010)

[49] Ejgenberg, Y., Farbstein, M., Levy, M., Lindell, Y.: SCAPI: The secure computation application programming interface. IACR Cryptology ePrint Archive (2012) 629

[50] Rastogi, A., Hammer, M.A., Hicks, M.: Wysteria: A programming language for generic, mixed-mode multiparty computations. In: Proceedings of the IEEE Symposium on Security and Privacy (Oakland). (2014)

[51] Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: CRYPTO. Volume 2729 of Lecture Notes in Computer Science., Springer (2003) 247–264

[52] Toft, T.: Secure data structures based on multi-party computation. In: PODC, ACM (2011) 291–292

[53] Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: EUROCRYPT. Volume 6632 of LNCS., Springer (2011) 169–188

[54] Damgård, I., Keller, M., Larraia, E., Miles, C., Smart, N.: Implementing AES via an actively/covertly secure dishonest-majority MPC protocol. In: Security and Cryptography for Networks. Volume 7485 of LNCS., Springer (2012) 241–263

[55] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO. Volume 7417 of LNCS., Springer (2012) 643–662

[56] Reistad, T.I., Toft, T.: Secret sharing comparison by transformation and rotation. In: Information Theoretic Security. Springer (2009) 169–180

[57] Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: TCC. Volume 3876 of Lecture Notes in Computer Science., Springer (2006) 285–304

[58] Toft, T.: Solving linear programs using multiparty computation. In: Financial Cryptography. Volume 5628 of LNCS., Springer (2009) 90–107

[59] Launchbury, J., Diatchki, I.S., DuBuisson, T., Adams-Moran, A.: Efficient lookup-table protocol in secure multiparty computation. In: International Conference on Functional Programming, ACM (2012) 189–200

[60] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)

[61] Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack problems. Springer (2004)

[62] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)

[63] Hoare, C.A.R.: Quicksort. Comput. J. **5**(1) (1962) 10–15

[64] Sedgewick, R.: Quicksort. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York (1975)

[65] Sedgewick, R.: Implementing quicksort programs. Communications of the ACM **21**(10) (1978) 847–857

[66] Williams, J.: Algorithm 232 - heapsort. Communications of the ACM **7**(6) (1964) 347–348

[67] Floyd, R.W.: Algorithm 245 - treesort. Communications of the ACM **7**(12) (1964) 701

[68] Knuth, D.E.: The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley (1973)

[69] Batcher, K.E.: A new internal sorting method. Technical Report GER-11759, Goodyear Aerospace (1964)

[70] Ajtai, M., Komlós, J., Szemerédi, E.: An O(n log n) sorting network. In: STOC, ACM (1983) 1–9

[71] Parberry, I.: The pairwise sorting network. Parallel Processing Letters **2** (1992) 205–211

[72] Goodrich, M.T.: Randomized shellsort: A simple data-oblivious sorting algorithm. J. ACM **58**(6) (2011) 27

[73] Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. IACR Cryptology ePrint Archive (2011) 122

[74] Zhang, B.: Generic constant-round oblivious sorting algorithm for MPC. In: ProvSec. Volume 6980 of Lecture Notes in Computer Science., Springer (2011) 240–256

[75] Arulanandham, J.J., Calude, C., Dinneen, M.J.: Bead-sort: A natural sorting algorithm. Bulletin of the EATCS **76** (2002) 153–161

[76] Arulanandham, J.J., Calude, C., Dinneen, M.J.: A fast natural algorithm for searching. Theor. Comput. Sci. **320**(1) (2004) 3–13

[77] Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: Practically efficient multi-party sorting protocols from comparison sort algorithms. In: ICISC. Volume 7839 of Lecture Notes in Computer Science., Springer (2012) 202–216

[78] Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. IACR Cryptology ePrint Archive (2014) 121

[79] Bogdanov, D., Laur, S., Talviste, R.: Oblivious sorting of secret-shared data. Technical Report T-4-19, Cybernetica, Institute of Information Security (2013)

[80] Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computing Conference. Volume 32 of AFIPS Conference Proceedings., Thomson Book Company, Washington D.C. (1968) 307–314

[81] Reistad, T., Toft, T.: Linear, constant-rounds bit-decomposition. In: ICISC. Volume 5984 of Lecture Notes in Computer Science., Springer (2009) 245–257

[82] Brams, S.J., Taylor, A.D.: Fair division - from cake-cutting to dispute resolution. Cambridge University Press (1996)

[83] Brams, S.J., Fishburn, P.C.: Fair division of indivisible items between two people with identical preferences: Envy-freeness, pareto-optimality, and equity. Social Choice and Welfare **17**(2) (2000) 247–267

[84] Jones, M.A.: Equitable, envy-free, and efficient cake cutting for two people and its application to divisible goods. Mathematics Magazine **75**(4) (2002) pp. 275–283

[85] Brams, S.J., Jones, M.A., Klamler, C.: Better ways to cut a cake - revisited. In: Fair Division. Volume 07261 of Dagstuhl Seminar Proceedings., Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2007)

[86] Algesheimer, J., Camenisch, J., Shoup, V.: Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In: Advances in Cryptology - CRYPTO 2002. Volume 2442 of Lecture Notes in Computer Science., Springer (2002) 417–432

[87] Brams, S.J., Taylor, A.D.: An envy-free cake division protocol. In: The American Mathematical Monthly. Volume 102., Mathematical Association of America (1995) 9–18

[88] Brams, S.J., Taylor, A.D., Zwicker, W.: A moving-knife solution to the four-person envy-free cake-division problem. In: The American Mathematical Monthly. Volume 125., Mathematical Association of America (1997) 547–554

[89] Dodis, Y., Halevi, S., Rabin, T.: A cryptographic solution to a game theoretic problem. In: Advances in Cryptology - CRYPTO 2000. Volume 1880., Springer (2000) 112–130

[90] Dodis, Y., Rabin, T.: Cryptography and game theory. Algorithmic Game Theory (2007) 181–207

[91] Izmalkov, S., Micali, S., Lepinski, M.: Rational secure computation and ideal mechanism design. In: 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), IEEE Computer Society (2005) 585–595

[92] Katz, J.: Bridging game theory and cryptography: Recent results and future directions. In: Theory of Cryptography, Fifth Theory of Cryptography Conference, TCC 2008. Volume 4948 of Lecture Notes in Computer Science., Springer (2008) 251–272

[93] Turing, A.M.: Rounding-off errors in matrix processes. The Quarterly Journal of Mechanics and Applied Mathematics **1**(1) (1948) 287–308

[94] Bunch, J.R., Hopcroft, J.E.: Triangular factorization and inversion by fast matrix multiplication. Mathematics of Computation **28** (1974) 231–236

[95] Horn, R.A., Johnson, C.R.: Matrix analysis. Cambridge University Press (1990)

[96] Su, F.E.: Rental harmony: Sperner's lemma in fair division. In: The American Mathematical Monthly. Volume 106., Mathematical Association of America (1999) 930–942

[97] Stromquist, W.: Envy-free cake divisions cannot be found by finite protocols. Electr. J. Comb. **15**(1) (2008)

[98] Halpern, J.Y., Teague, V.: Rational secret sharing and multiparty computation: extended abstract. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing, ACM (2004) 623–632

[99] Gordon, S.D., Katz, J.: Rational secret sharing, revisited. In: Security and Cryptography for Networks, 5th International Conference, SCN 2006. Volume 4116 of Lecture Notes in Computer Science., Springer (2006) 229–241

[100] Nielsen, J.B., Alwen, J., Cachin, C., Nielsen, J.B., Pereira, O., Sadeghi, A.R., Schomakers, B., Shelat, A., Visconti, I.: Summary report on rational cryptographic protocols (2007)

[101] Garay, J.A., Katz, J., Maurer, U., Tackmann, B., Zikas, V.: Rational protocol design: Cryptography against incentive-driven adversaries. In: 54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA. (2013) 648–657

[102] Kocher, P.C.: Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. In: CRYPTO. Volume 1109 of Lecture Notes in Computer Science., Springer (1996) 104–113

[103] Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3) (June 1983) 362–391

[104] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3) (1987) 596–615

[105] Toft, T.: Primitives and Applications for Multi-party Computation. PhD thesis, Department of Computer Science, Aarhus University (2007)

[106] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271

[107] Sedgewick, R., Vitter, J.S.: Shortest paths in euclidean graphs. Algorithmica **1**(1) (1986) 31–48

[108] Du, W., Atallah, M.J.: Secure multi-party computation problems and their applications: a review and open problems. In: NSPW. (2001) 13–22

[109] Kruger, L., Jha, S., Goh, E.J., Boneh, D.: Secure function evaluation with ordered binary decision diagrams. In: ACM CCS, ACM (2006) 410–420

[110] Ishai, Y., Paskin, A.: Evaluating branching programs on encrypted data. In: Theory of Cryptography, TCC 2007. Volume 4392 of LNCS., Springer (2007) 575–594

[111] Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: ACM CCS. CCS '07, ACM (2007) 498–507

[112] Barni, M., Failla, P., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Privacy-preserving ECG classification with branching programs and neural networks. IEEE TIFS **6**(2) (June 2011) 452–468

[113] Brickell, J., Shmatikov, V.: Privacy-preserving graph algorithms in the semi-honest model. In: ASIACRYPT. Volume 3788 of LNCS. Springer (2005) 236–252

[114] Knuth, D.E.: The Art of Computer Programming, Volume I: Fundamental Algorithms. Addison-Wesley (1968)

[115] Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. IACR Cryptology ePrint Archive (2014) 137

[116] Edmonds, J.: The chinese postman problem. Operations Research **13** (1965)

[117] West, D.B.: Introduction to Graph Theory. Prentice Hall (2000)

[118] Edmonds, J., Johnson, E.L.: Matching, euler tours and the chinese postman. Mathematical Programming **5**(1) (1973) 88–124

[119] Biggs, N., Lloyd, E.K., Wilson, R.J.: Graph Theory, 1736-1936. Oxford University Press (1986)

[120] Bondy, J.A., Murty, U.S.R.: Graph theory. Graduate texts in mathematics. Springer (2007)

[121] Bellman, R.: On a routing problem. Quarterly of Applied Mathematics (16) (1958) 87–90

[122] Chen, M., Chowdhury, R.A., Ramachandran, V., Roche, D.L., Tong, L.: Priority queues and dijkstra's algorithm. Technical Report TR-07-54, UTCS (2007)

[123] Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. Canadian Journal of Mathematics **8** (1956) 399–404

[124] Ford, L.R., Fulkerson, D.R.: Flows in networks. Princeton University Press (1962)

[125] Dinic, E.A.: Algorithm for solution of a problem of maximum flow in a network with power estimation. Soviet Math. Doklady **11** (1970) 1277–1280

[126] Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. Journal of the ACM **19**(2) (1972) 248–264

[127] Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum flow problem. In: Proceedings of the eighteenth annual ACM symposium on Theory of computing. STOC '86, ACM (1986) 136–146

[128] Bar-Ilan, J., Beaver, D.: Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, ACM (1989) 201–209

[129] Osborne, M.J., Rubinstein, A.: A course in game theory. MIT press (1994)

[130] Osborne, M.J.: An Introduction to Game Theory. Oxford University Press (2004)